

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind

MODELLIERUNG VON FEHLVERHALTEN MIT BERÜCKSICHTIGUNG PARALLELER ABLÄUFE

Karl Weber

*Diss. Erlangen
1983*

DISSERTATION

BAND 16 · NUMMER 9 · ERLANGEN · AUGUST 1983

1983 № 346



ISSN 0344 - 3515



Digitized by the Internet Archive
in 2026

MODELLIERUNG VON FEHLVERHALTEN MIT BERÜCKSICHTIGUNG PARALLELER ABLÄUFE

Der Technischen Fakultät
der Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR - INGENIEUR

vorgelegt

von

Karl Weber

Erlangen 1983

Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung: ⁹ 11. 5. 1983

Tag der Promotion: 21. 6. 1983

Dekan: Prof. Dr. Zwicker

Berichterstatter: Prof. Dr. Hofmann

Prof. Dr. Wedekind

A b s t r a c t

MODELLIERUNG VON FEHLVERHALTEN MIT BERÜCKSICHTIGUNG PARALLELER ABLÄUFE:

In dieser Arbeit wird ein Modell zur Behandlung von Fehlverhalten von sequentiellen und asynchronen Abläufen entwickelt.

Nach einer informellen Erläuterung des Systemverhaltens bei normaler und fehlerhafter Abwicklung wird ein allgemeines Modell entwickelt, mit dessen Hilfe man konsistente Zustände nach Defekten erreichen kann. Danach wird der Domino-Begriff näher untersucht und präzisiert. Damit können notwendige und hinreichende Bedingungen zur Erstellung dominofreier Systeme angegeben werden. Sowohl diese Definition als auch die daraus abgeleiteten Regeln sind allgemeinerer Natur als die aus der Literatur bekannten Darstellungen. Weiterhin werden Sprachkonstrukte zur Spezifikation von Fehlerbehandlungsproblemen entwickelt.

Es läßt sich zeigen, daß die hier vorgestellte Methode auf adäquate Weise auch für verteilte Systeme angewandt werden kann.

Inhaltsverzeichnis

0.	Einleitung und Überblick	1
1.	Erläuterung zentraler Begriffe	8
1.1	System	8
1.2	Prozesse	10
1.3	Atomare Aktionen	12
1.4	Zuverlässigkeit, Korrektheit, Defekt und Fehler	15
1.5	Fehlertoleranz	17
1.6	Phasen der Fehlerbehandlung	19
2.	Fehlererkennung	21
2.1	Fehlerquellen	22
2.2	Prüfstellen	28
2.3	Erkennungsmethoden	29
2.4	Diagnose	34
3.	Fehlermaskierung und -eindämmung	40
3.1	Fehlerbeseitigung auf Hardwareebene	40
3.2	Fehlereindämmung zum Schutz vor Softwaredefekten	46
4.	Fehlerbehandlung im Software- Entwicklungsprozeß	51
4.1	Software-Entwicklungsprozeß	55
4.2	Methoden zur Verbesserung der Qualität während des Entwicklungszyklus	62
4.3	Recovery-Blöcke	64
4.4	N-Version-Programmierung	68
4.5	Bewertung der einzelnen Verfahren	70

5.	Recovery-Arten	71
5.1	Unerwünschte Ereignisse	73
5.2	Rückwärts-Recovery	79
5.3	Recovery und parallele Aktivitäten	84
6.	Das Transaktionskonzept	90
6.1	Synchronisation	93
6.2	Protokolle für die Verwaltung der Recovery-Daten	96
6.3	Abschluß einer Transaktion	100
7.	Modellierung von parallelen Berechnungen mit Fehlerbehandlung	103
7.1	Zielvorstellungen	103
7.2	Ein Recovery-Modell	108
7.3	Fehlerausbreitung	115
7.4	Der Domino-Effekt	125
7.5	Sprachliche Darstellungsform	135
8.	Recovery-Modelle für Prozeßsysteme	143
8.1	Recovery in Nachrichtensystemen	144
8.2	Ereignisgraphen	151
8.3	Monitor-Recovery	156
9.	Verteilte Systeme	163
9.1	Anforderungen an das Recovery-Konzept	166
9.2	Vereinfachtes Modell	169
9.3	Behandlung der Recovery-Daten	178
9.4	Beispiel-System	183
10.	Zusammenfassung und Schlußbemerkungen	187
11.	Literaturhinweise	190

0. Einleitung und Überblick

Mit dem wachsenden Einsatz computergesteuerter Systeme zur Überwachung technischer Anlagen, die Risiken für Leben und Gesundheit von Menschen in sich bergen und die Umweltschäden hervorrufen können, steigt auch die Forderung nach Zuverlässigkeit und Sicherheit. Auch kommerzielle Anwendungen in Banken, Versicherungen und Behörden erfordern fehlersichere Systeme, um aufwendige manuelle Nachprüfungen abbauen zu können.

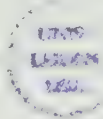
Der Alltag in der Softwareentwicklung kann diesen Anforderungen nur selten gerecht werden. Meist fällt es schwer, die komplexen Aufgaben in der vorgegebenen Zeit abzuwickeln. Überlegungen zur Zuverlässigkeit werden deshalb oft zurückgestellt. Auftretende Mängel versucht man nachträglich zu beseitigen. Bei der Softwareerstellung unter Termindruck geht aber oft die Übersichtlichkeit verloren, so daß die Modifikation und Wartung der Programme hohe Kosten verursachen. So stellt der Versuch, nachträglich die geforderte Qualität des Produktes zu erreichen, den Nutzen des gesamten Projektes in Frage.

Durch modularen Aufbau, formale Spezifikation, höhere Programmiersprachen und Verifikation versuchte man die Zuverlässigkeit von Programmen zu erhöhen. Diese Methoden und Werkzeuge allein können nicht garantieren, daß damit entwickelte Systeme mit einer hohen Wahrscheinlichkeit das gewünschte Verhalten zeigen.

Nur wenn man von Anfang an berücksichtigt, daß

* Software auf fehlerbehafteter Hardware ablaufen muß,

* keine Methode und kein Werkzeug perfekt ist (selbst



bei Verifikation kann nicht ausgeschlossen werden, daß es zu Fehlinterpretationen oder Versagen des unterstützenden Automaten kommt),

- * die betrachteten Systeme selten vollständig abgeschlossen sind (es gibt Einflüsse, die nicht der Kontrolle einer fest umrissenen Anwendung unterliegen),

- * bei komplexen Gebilden die Gefahr des Ausfalls groß ist, selbst wenn die Wahrscheinlichkeit eines Defektes einzelner Komponenten gering ist,

lassen sich Strukturen finden, die auch hohen Zuverlässigkeitsanforderungen genügen.

Das resultierende System wird dann hoffentlich den Bedürfnissen seiner Benutzer gerecht werden. Dies ist auch ein Maßstab für Zuverlässigkeit. Verfügbarkeit läßt sich durch den Quotienten der Reparaturzeit zur Lebensdauer des Systems quantifizieren.

Es sei angemerkt, daß der von Verifikationstheorie geprägte Begriff der "Korrektheit" nicht in Konflikt zu der das Betriebsverhalten charakterisierenden "Zuverlässigkeit" steht. Beide Begriffe ergänzen einander. Ein System, das den Anforderungen der Benutzer genügen soll, läßt sich nicht aufbauen auf einer Software, die unstrukturiert und ungeprüft in Betrieb genommen wird. Andererseits sind ohne das Bestreben nach Zuverlässigkeit korrekte Programme wertlos.

Ähnlich verhält es sich mit dem Zusammenhang zwischen Leistungsfähigkeit und Zuverlässigkeit. Hohe Durchsatzraten sind sinnlos, wenn man nicht auf die Richtigkeit der Ergebnisse vertrauen kann. Hohe Verfügbarkeit ist wertlos, wenn die benötigte Verarbeitungsleistung nicht erbracht werden kann.

Unter Berücksichtigung dieser Abhängigkeiten müssen Methoden und Werkzeuge gefunden werden, mit denen sich

verschiedene Arten von negativen Einflüssen auf das System behandeln lassen. Dadurch darf aber die Komplexität der Lösungen nicht zu stark wachsen.

Doch erhöhte Zuverlässigkeit und Verfügbarkeit verursacht Kosten - beim Entwurf wie im Betrieb. Wie man den Aufwand dafür klein hält und trotzdem den gewünschten Erfolg erzielen kann, soll auch in den folgenden Abschnitten untersucht werden.

Zuerst wird versucht, die Kernbegriffe informell zu erläutern, um dem Leser einen Einblick in die verwendete Terminologie zu geben. Die Begriffe werden relativ allgemein definiert, um sie später auf möglichst viele existierende Modelle anwenden zu können.

Anschließend wird die Frage untersucht, wie man Fehler erkennen kann, wie die Ursachen aufgedeckt werden und wie sie sich einordnen lassen. Fehlererkennung ist der erste Schritt zur Behebung. Untersuchungen über Fehlerquellen und Methoden zu deren Aufdeckung im Entwurf und im Betrieb sind als Grundlage unverzichtbar. Ohne sie ist eine erfolgreiche Fehlerbehandlung nicht durchführbar.

Um das System in einen funktionsfähigen Zustand überführen zu können, ist neben der Erkennung eines Defektes oft auch eine Diagnose wichtig. Erst dann lassen sich die fehlerhaften Komponenten reparieren oder austauschen. Danach werden Konzepte vorgestellt, wie man Folgeschäden eines Fehlers eingrenzt.

Da man aber in vielen Fällen einen Defekt nicht sofort erkennen kann, zieht eine falsch durchgeführte Funktion auch noch intakte Komponenten in Mitleidenenschaft. Es ist deshalb ein Systemzustand zu rekonstruieren, der vor dem Fehler existiert hat oder existiert haben könnte.

Auch die Entwicklung von Software erfordert den Einsatz von Techniken, die helfen sollen, möglichst rasch Fehler zu identifizieren. Eine Reihe von Methoden wird

angesprochen, die zum Auffinden von Defekten vor oder während der Inbetriebnahme dienen.

Für die Behandlung von Programm-Entwurfsfehlern im Betrieb genügt es nicht, nur einen konsistenten Zustand wiederherzustellen. Eine Wiederholung desselben Ablaufs führt dann wieder zu denselben Fehlern. Man muß also erreichen, daß bei folgenden Berechnungen dieser Defekt umgangen wird.

Eine Möglichkeit, Entwurfsfehler auszuschalten, bieten das Recovery-Block-Konzept und die N-Version-Programmierung. Beide Methoden gehen davon aus, daß mehrere Programmierteams ein Problem lösen. Beim Recovery-Block-Konzept bearbeitet man immer erst eine Alternative. Nur wenn der Abschlußtest negativ ausfällt, wird danach der Zustand vor dem Durchlauf wiederhergestellt und eine Alternativfunktion ausgeführt.

Wesentlich diversitärer geht man bei der N-Version-Programmierung vor. Hier werden neben den verschiedenen Teams auch unterschiedliche Programmiersprachen, Systemsoftware und Hardwarekomponenten zur Lösung des Problems eingesetzt. Auf diese Weise schafft man zwei völlig disjunkte Pfade, die nur die Anforderungsdefinition und Vergleiche der Resultate mit Hilfe von Hardwareeinrichtungen gemeinsam haben.

Klassifiziert man Fehler genauer, die bei Rechnersystemen auftreten, so entdeckt man eine Gruppe von Fehlern, die "kalkulierbar", aber nur sehr schwer vermeidbar sind. Hierunter fallen vor allem viele kurzfristige Hardwarestörungen, aber auch Sonderfälle im Programmfluß, wie "Dateiende erreicht" oder "Division durch Null". Diese Ereignisse lassen sich nicht in den normalen Ablauf integrieren. Wenn möglich, sollte man sie von den gewöhnlichen Operationenfolgen trennen, um sie mit separaten "Handlern" besser in den Griff zu bekommen.

Schwierig gestaltet sich bei der Rekonstruktion eines

konsistenten Systemzustandes das Zurücksetzen von asynchronen Prozessen, die gemeinsame Betriebsmittel benutzen oder miteinander kommunizieren. Probleme, die hier auftreten, stehen in enger Beziehung zu Synchronisationsfragen.

Wie sich nicht aufeinander abgestimmte Sicherungs- und Synchronisationsmaßnahmen auswirken, läßt sich wohl am besten anhand des Domino-Effektes veranschaulichen. Der Domino-Effekt charakterisiert den Fall, bei dem trotz Vorsorgemaßnahmen bestimmte Fehler einen Neustart des Systems notwendig machen.

In den Abschnitten 6 bis 9 werden die Zusammenhänge zwischen Synchronisation und der Schadensbeseitigung (Recovery) näher untersucht.

Zuerst wird der Ansatz erläutert, der sich bei Datenbank Anwendungen durchgesetzt hat, um die Konsistenz der gesammelten Information zu wahren und den Domino-Effekt zu vermeiden. Hier belegt man die einzelnen Sätze (oder andere logische Abschnitte) so, daß eine zu einer seriellen Durchführung äquivalente Ablauffolge von Aufträgen entsteht. Diese Aufträge werden Transaktionen genannt. Sie sorgen dafür, daß die Datenbank von einem logisch konsistenten Zustand ausgehend wieder einen konsistenten Zustand erreicht. Transaktionen müssen entweder erfolgreich abgeschlossen werden oder dürfen keine Rückwirkungen auf das System haben. Es darf keine Interferenzen zwischen laufenden Transaktionen geben. Einziges Ziel dieser Maßnahmen ist, die Datenbank konsistent zu halten.

In Kapitel 7 wird versucht, ein allgemeines Konzept für Recovery-Aktivitäten zu erarbeiten. Dazu wird zuerst die Zielvorstellung herausgearbeitet, die als Maßstab für das Recovery-Verhalten dienen soll. Für einige Anwendungen lassen sich diese Ideale direkt übertragen. Doch auch in diesen Fällen wird bereits das Prinzip deutlich, das bei der Recovery dominiert.

Durch Zurückkehren zu einem konsistenten Zustand versucht man die Basis für das weitere Vorgehen zu schaffen. Ein Modell wird vorgestellt, das die Abhängigkeit von Aktionen aufzeigt. Wenn eine Operation eine andere beeinflusst, so kann dies Auswirkungen auf das Recovery-Verhalten haben. Einschränken läßt sich diese Abhängigkeit durch die Validierungsrelation, die feststellen hilft, ob eine Änderung bereits akzeptiert worden ist.

Man kann ein Regelwerk angeben, das den Rücksetzvorgang steuert. Mit diesem Modell läßt sich formal festlegen, wann ein System domino-frei ist. Diese Definition erlaubt es allerdings nicht, auf einfache Art und Weise zu entscheiden, ob die Bedingungen nun auch wirklich erfüllt sind. Deshalb wird noch eine weitere Formulierung angeboten, die eine lokale Betrachtungsweise besser unterstützt. Den Abschluß dieses Kapitels bilden Überlegungen zur Spezifikation des Recovery-Verhaltens.

Auch andere Modelle werden herangezogen und miteinander verglichen. Der Nachrichtenmechanismus zeichnet sich durch besonders gute Fehlerbehandlungseigenschaften aus. Es gibt keine gemeinsamen Datenbereiche; Informationen zwischen den Prozessen werden nur explizit ausgetauscht. Läßt man eine Fehlerausbreitung vom Empfänger auf den Sender nicht zu und schränkt den Aufbau von Recovery-Blöcken entsprechend ein, so sind die resultierenden Systeme domino-frei.

Durch Ereignisgraphen lassen sich dynamische Abläufe beschreiben. Hier sind weitere Recovery-Möglichkeiten erlaubt. Es existiert ein Protokoll, das den Ablauf der Fehlerbehandlung steuert.

Ein System, dem das Monitor-Konzept zugrunde liegt, wird ebenfalls behandelt. Dabei zeigt sich, wie problematisch es ist, vernünftige Bedingungen zu finden, die den Aufbau domino-freier Systeme ermöglichen.

Verteilte Systeme treten immer stärker in den Vordergrund - gerade auch wegen der potentiell günstigeren Aussichten, bei Fehlern noch weiterarbeiten zu können. Recovery-Mechanismen müssen auf diese Struktur Rücksicht nehmen. Das Zusammenwirken von Synchronisation und Fehlerbehandlung wird dann für einfache Anwendungen skizziert. Dabei muß auch darauf eingegangen werden, wie man Recovery-Daten behandelt, um eine Fehlerbehandlung bei Ausfällen einzelner Knoten zu ermöglichen. Beispiele vertiefen diese Überlegungen.

Diese Arbeit liefert den Rahmen für konkrete Anwendungen. Praktische Probleme auf manchen Gebieten der Fehlerbehandlung lassen sich mit Hilfe dieser allgemeinen Überlegungen besser in den Griff bekommen. Vor allem wird der Einfluß von Synchronisation und Systemaufbau bei der Erkennung und Behebung von Defekten dargestellt.

1. Erläuterung zentraler Begriffe

In der Einleitung wurde bereits eine Reihe von Ausdrücken aus dem Gebiet der Fehlerbehandlung verwendet, ohne auf deren Bedeutung einzugehen. Hier wird nun die verwendete Terminologie erläutert. Sie soll sowohl auf Hardware- wie auch auf Softwarekomponenten anwendbar sein.

1.1 System

Der Name kommt aus dem Griechischen und läßt sich mit "Zusammenstellung" übersetzen. Er taucht sowohl in der technischen wie auch in der sozialen Welt auf. Auf diesen Kernbegriff beziehen sich die nachfolgenden Termini.

Da eine große Menge von Systemen und deren Abbildung in abstrakte Modelle (auch das sind Systeme) in den Informatik-Fachgebieten entworfen werden mußten, kann eine nur recht allgemeine Erläuterung an dieser Stelle angeboten werden. In anderen Wissenschaften (z.B. Chemie) hat sich ein etwas spezifischerer Systembegriff durchgesetzt.

Ein System besteht aus einer Menge von Komponenten, die auf eine bestimmte Art und Weise miteinander zusammenarbeiten.

Komponenten oder Elemente sind gekennzeichnet durch ihre internen Zustände.

Durch die Übergangsrelation wird angegeben, wie die Komponenten aufeinander einwirken und wie externe Ereignisse auf interne Zustandsänderungen abgebildet werden.

Komponenten lassen sich weiter zerlegen. Die einzel-

nen Elemente können auch zu mehreren Systemen gehören.

Diese Definition von "System" umfaßt dynamische, diskrete, nichtdeterministische Anordnungen.

Statische Systeme, die keine Zustandsänderungen durchführen (wie das Periodensystem), findet man in der realen Welt kaum.

Kontinuierliche Systeme wie der Energiefluß spielen zwar eine große Rolle in der Natur und in der technischen Welt, die Rechenanlagen zerlegen sie aber in diskrete Zusammenstellungen. Fließende Zustandsänderungen, mathematisch beschrieben durch Differentialgleichungen, werden von der Regelungstechnik und verwandten Fachrichtungen behandelt.

Prozessoren bearbeiten die einzelnen Programme vorwiegend deterministisch. Jedoch sind sie in eine Umgebung eingebettet, in der es zwischen den einzelnen Aktivitäten in einem Zeitraum oft keine festgelegte Reihenfolge und keine spürbaren Zusammenhänge gibt. Fehler sind nicht vorhersehbar und deshalb wohl kaum mit einem deterministischen Modell sinnvoll zu analysieren.

Der Systemzustand setzt sich aus den Zuständen der Komponenten zusammen.

Systeme können, wie gesagt, wiederum zusammengefaßt werden. Die Umgebung eines Systems ist die Menge der Systeme, die es beeinflussen können oder auf die es selbst einwirkt.

Die statische Struktur wird durch die Übergangsrelation beschrieben. Die statische Struktur ist maßgebend für die Arten von Fehlern. Die Effekte der Fehler werden durch die dynamische Struktur dargestellt. Die Dynamik läßt sich durch die Folge von Zuständen repräsentieren, die Aktivitäten hervorrufen. Sie werden produziert durch Änderungen von Komponenten des Systems.

Die Begriffe "Aktivität", "Aktion" und "Operation" sind synonym. Sie kennzeichnen Zustandsübergänge, ohne darauf einzugehen, wodurch diese hervorgerufen werden.

1.2 Prozesse

Die Aktivitäten eines Systems lassen sich auf vielerlei Weise betrachten, je nachdem, welche Aspekte betont und welche vernachlässigt werden. Eine einfache und weit verbreitete Art, die Dynamik von Rechenanlagen darzustellen, ist das Prozeßkonzept. Abhängig vom Standpunkt des Benutzers sind verschiedene Prozesse und deren Interaktionen bestimmend für die Durchführung der Aufgaben.

Die Abwicklung von Benutzeraktivitäten läßt sich formal wie folgt darstellen (/Hof76/).

Ein Prozeß ist ein Tripel $P=(Z,R,S)$

wobei

- $Z=V_1 \times V_2 \times \dots \times V_l$ die Zustandsmenge ist

($V_i = V_i \cup \{*\}$, $* \notin V_i$ für $i = 1 \dots l$)

- $S \in Z$ die Menge der Anfangszustände

- $R \in Z \times Z$ die Übergangsrelation

Ein "*" in der ersten Komponente eines Elementes von R bedeutet, daß die entsprechende Variable nicht abgefragt wird. Ein "*" in der zweiten Komponente gibt an, daß der Wert der Variablen nicht verändert wird.

Sei $PS=(Z^*, S^*, R^*)$ ein aus $P_1, \dots, P_m$ zusammengesetztes Prozeßsystem.

Dabei gilt:

- 1) Es existiert eine Funktion f

$$f : \{(1,1), \dots, (1,n1), \dots, (n,nm)\} \rightarrow \{1, \dots, k\}$$

f hat die Aufgabe, die einzelnen Prozeßvariablen auf V^* abzubilden.

- 2) $Z^* = V1 \times \dots \times V_k$ die Menge der Zustände.

Es gelten dieselben Anforderungen wie für P .

- 3) $AB(z_i, z, i)$ ist ein Prädikat, das angibt, ob ein Zustand von P_i auf einen Zustand von P^* "abbildbar" ist:

$$z_i = (v_{i,1}, \dots, v_{i,n_i}) \in Z_i, z = (v_1, \dots, v_k) \in Z^*$$

$$v_{i,j} \neq * \wedge f(i,j) = n \rightarrow v_{i,j} = v_n \text{ für alle } j = 1, \dots, n_i$$

Mit diesem Prädikat läßt sich angeben, ob sich ein Zustandsvektor eines Prozesses in ein Element der Zustandsmenge des Prozeßsystems einbetten läßt.

- 4) $S^* \in Z^*$ ist die Menge der Anfangszustände

(Für jedes s aus S^* existieren $s_i \in S_i$ mit $AB(s_i, s, i)$ für alle $i = 1, \dots, m$).

- 5) $R^* \in Z^* \times Z^*$ ist die Übergangsrelation

(Für jedes (z, z') aus R^* existiert eine Menge T von Tupeln (z_i, z'_i) aus verschiedenen R_i für die gilt $AB(z_i, z, i)$ und $AB(z'_i, z', i)$. Wenn ein v_n aus $z \neq *$, so gibt es ein $v_{i,j} = v_n$ aus einem der z_i und es ist $f(i,j) = n$ (analoges gilt für ein $v_{n'}$ aus z')).

Damit wird besonders das Konzept der gemeinsamen Variablen hervorgehoben, das beim Betriebssystementwurf eine große Rolle spielt. Es ist aber nur mit Vorbehalten möglich, verteilte Systeme damit zu modellieren.

1.3 Atomare Aktionen

Der Begriff "atomar" wird hier in einer leicht gegenüber dem normalen Sprachgebrauch modifizierten Art angewandt.

Er wird dazu benutzt, um mögliche Auswirkungen von Fehlern besser erkennen zu können.

Eine Aktion ist eine zusammengehörende Menge von Zustandsübergängen (Transitionen).

Eine Aktion wird als atomar angesehen, wenn es zwischen den Prozessen, die an dieser Aktion beteiligt sind, und den anderen Aktivitäten des Systems nur "zyklenfreien" Informationsfluß gibt.

Man kann atomare Aktionen schachteln, aber sie dürfen sich nicht überlappen.

Unerwünschte Interferenzen zwischen den Transitionen verschiedener atomarer Aktionen werden häufig durch gegenseitigen Ausschluß vermieden. Eine andere Eigenschaft, die Mengen von atomaren Aktionen aufweisen, ist die Serialisierbarkeit. Kann man für alle möglichen Abwicklungen der Transitionen sequentielle Folgen der Aktionen angeben, die dasselbe Ergebnis liefern würden, so sind alle beteiligten Aktionen atomar.

Es gibt verschiedene Anschauungen darüber, unter welchen Bedingungen zwei Abläufe als äquivalent angesehen werden können. Sie müssen nicht nur zum selben Endzustand führen, sondern zusätzlich dazu noch bestimmte Beziehungen zwischen den Transitionen aufrechterhalten (/Esw76/).

Alle diese Erläuterungen veranschaulichen zwar den Begriff, liefern aber keine exakte Darstellung des Sachverhalts.

Als Modell für eine formale Interpretation von "atomarer Aktion" sollen Ereignisgraphen dienen.

Ein Ereignisgraph $G = (E, B)$ besteht aus einer nicht-leeren Menge E von Knoten und einer Menge $B \subseteq E \times E$ von

gerichteten Kanten über E (/Bes81/).

Elemente aus E kann man als Ereignisse (oder Aktionen) auffassen, die im Verlauf einer Berechnung auftreten. Die gerichteten Kanten beschreiben Abhängigkeiten von Ereignissen. (e_1, e_2) kann so interpretiert werden, daß e_1 zuerst eine Variable v verändert und e_2 diese Größe nachfolgend liest oder modifiziert. Ereignis e_1 muß also vor e_2 stattfinden.

Aus den oben angeführten Anschauungen läßt sich die Forderung ableiten, daß Ereignisgraphen zyklensfrei sein sollen. Ein Zyklus würde bedeuten, daß Ereignis e_1 auf e_2 folgt und umgekehrt. Sollen nun mehrere Transitionen zu einer komplexeren Aktion zusammengefaßt werden, so läßt sich das auch im Ereignisgraphen nachvollziehen. Dieses Verfahren nennt man "Collapsing".

Ein Ereignisgraph $G/A/ = (E/A/, B/A/)$ geht durch "Collapsing" aus $G = (E, B)$ hervor, wenn gilt:

$o(A)$ ist eine Teilmenge von E , A ist nicht in E enthalten

$E/A/$ enthält $E - o(A)$, die Restmenge von E ohne $o(A)$, und A

$B/A/$ ist definiert durch

wenn e und e' aus $E - o(A)$, dann ist (e, e') in $B/A/$

wenn e aus $E - o(A)$ und e' aus $o(A)$, dann ist (e, A) in $B/A/$

wenn e aus $o(A)$ und e' aus $E - o(A)$, dann ist (A, e') in $B/A/$

(für alle (e, e') aus B)

Wenn durch "Collapsing" von G bzgl. A ein zyklensfreier Ereignisgraph G in ein $G/A/$ überführt wird, das ebenfalls keine Zyklen enthält, so sagt man, daß die Menge der Ereignisse $o(A)$ eine atomare Aktion bilden. Dies ist eine präzise Formulierung der Forderung, daß der Informationsfluß zwischen atomaren Aktionen zyklensfrei sein muß. Ist diese Eigenschaft erfüllt, so ist auch gewährleistet, daß die Ereignisse des Graphen $G/A/$ serialisierbar sind.

Der Aufwand, zum Feststellen, ob eine Teilmenge atomar auftritt, wächst an, wenn die Zahl der Knoten und Kanten im Ereignisgraphen größer wird. Deshalb sucht man nach Kriterien, die zwar schwächere Aussagen liefern, dafür aber den Aufwand für die Überprüfung der Eigenschaft verringern.

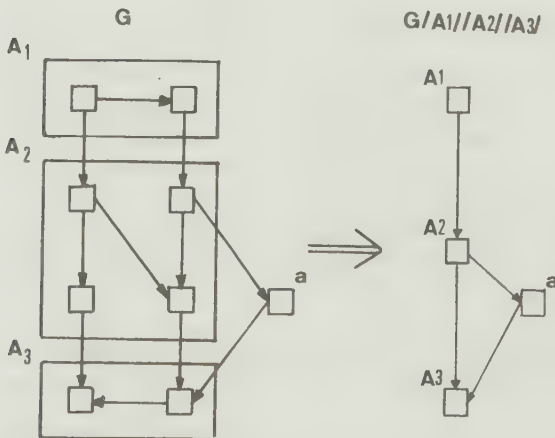


Abbildung 1.1:

Atomares Auftreten von Aktionen

Eine Kontraktion bezüglich A ist eine Teilmenge $o(A)$ von E, für die gilt:

wenn (e_1, A) und (A, e_2) in B/A enthalten sind, so existiert ein a aus $o(A)$, und es gibt in G einen Pfad von e_1 nach e_2 , der a als Zwischenknoten enthält.

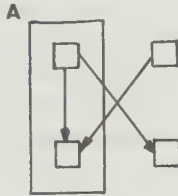


Abbildung 1.2:

Kontraktion und atomares Auftreten (A tritt atomar auf, ist aber keine Kontraktion)

Man kann zeigen, daß Kontraktionen atomar auftreten. Die Umkehrung gilt allerdings nicht, wie Abbildung 1.2 beweist.

Die Bedeutung von atomaren Aktionen liegt vor allem darin, daß damit einzelne Abschnitte in Prozeßsystemen wie sequentielle Programme behandelt werden können. Der Programmierer braucht keine Überlegungen anzustellen, wie die parallel dazu ablaufenden Aktivitäten durchgeführt werden. Außerdem lassen sich damit die Auswirkungen von Fehlern in vielen Fällen leichter bestimmen (/Lom77/).

1.4 Zuverlässigkeit, Korrektheit, Defekt und Fehler

Nachdem der Systembegriff untersucht worden ist, sollen nun die speziell für die Fehlerbehandlung wichtigen Begriffe näher erläutert werden. Wenn nötig,

werden auch die englischen Bezeichnungen mitangegeben, damit Querbezüge zu Veröffentlichungen möglich sind.

Ein Fehler (failure) ist die Abweichung des Systems von seinem spezifizierten Verhalten. Dabei ist nicht festgelegt, in welcher Form diese Spezifikation vorliegt.

Zuverlässigkeit ist ein Maß, wieweit das System seine Spezifikation erfüllt. Genügt es den Anforderungen mit einer hohen Wahrscheinlichkeit, so bezeichnet man ein System als zuverlässig (/Kop79/). Dabei können bestimmte Aktionen oder gewisse Zustände mehr oder weniger stark gewichtet werden.

Korrekt verhält sich ein Programm genau dann, wenn mit mathematischen Methoden beweisbar ist, daß sich das Eingangsprädikat P in das Ausgangsprädikat Q umformen läßt. Wie die Zusicherungen P und Q die Anforderungen der Benutzer widerspiegeln, bleibt dabei offen. Dabei geht man im allgemeinen davon aus, daß das Programmstück ohne Einfluß von außen durch ein idealisiertes fehlerfreies System ausgeführt wird.

Wie aus den Definitionen von Zuverlässigkeit und Korrektheit hervorgeht, ist es schwierig, einem System diese Prädikate zuzuordnen. Aber auch dann ist immer noch nicht ausgeschlossen, daß diese Anordnung fehlerfrei abläuft.

Für die Fehlerbehandlung sind diese Definitionen alleine nicht aussagekräftig genug. Nicht nur die Auswirkungen interessieren, sondern ebenso der Ort des Auftretens und der Umfang des dadurch entstandenen Schadens.

Auch da kann man auf das Modell aus 1.1 zurückgreifen. Als inkonsistent bezeichnet man einen Zustand, der aus einer nicht richtig durchgeführten Funktion resultiert (erroneous state). Im folgenden können unter Benutzung von korrekten Funktionen fehlerhafte Resul-

tate entstehen. Die Menge der inkonsistenten Objekte (error) enthält diejenigen Zustandsvariablen, die zur Abweichung vom spezifizierten Verhalten führen können, wenn sie in folgenden Aktionen benutzt werden. Ziel der Fehlerbehandlung muß es sein, diesen inkonsistenten Teil zu lokalisieren und zu korrigieren. Oft wird man sich damit begnügen, eine Obermenge der inkonsistenten Variablen zu betrachten.

Als Defekt (fault) wird die Ursache eines Fehlers bezeichnet.

Die Fehlerdichte ist das Verhältnis der defekten Komponenten zu deren Gesamtzahl. Die Fehlerrate gibt an, wie groß der Anteil der fehlerhaft ausgeführten Aktionen ist.

Verfügbarkeit a ist der Anteil der Betriebsdauer (MTBF) an der gesamten Missionszeit, die aus Reparaturzeiten (MTTR) und Einsatzintervallen besteht.

$$(a = \text{MTBF} / (\text{MTBF} + \text{MTTR}))$$

Fehler können eingeschleust werden durch Mängel in der Anforderungsanalyse, im Entwurf und in der Implementierung, durch Defekte von Komponenten während des Betriebs (aufgrund physikalischer Störungen) oder durch Fehlbedienungen.

Defekte durch physikalische Störungen werden im allgemeinen charakterisiert durch die Dauer (transient / permanent), das Ausmaß (lokal / verteilt) und den dabei erzeugten Zustand (determiniert / stochastisch).

Entwurfsfehler (Softwarefehler gehören hauptsächlich dieser Kategorie an) lassen sich nur schlecht in dieses Schema einordnen.

1.5 Fehlertoleranz

Man kann versuchen, Fehler beim Entwurf auszuschließen (Fehlerintoleranz oder -vermeidung) oder durch Einbau zusätzlicher Komponenten und Verbindungen ihre

Auswirkungen einzuschränken (Fehlertoleranz). Diese Unterscheidung wurde in /Avi75/ eingeführt.

In realen Systemen wird man oft zu einer Kombination greifen zwischen wenig flexiblen Entwurfsentscheidungen bei den Versuchen, Defekte zu vermeiden, und den relativ aufwendigen Absicherungen während der Durchführung, die bei Fehlertoleranztechniken notwendig werden.

Da Fehlertoleranztechniken auch zuverlässige Komponenten benötigen und andererseits jedes Modul nur eine beschränkte Verfügbarkeit besitzt, lassen sich diese beiden Techniken nur sehr schlecht voneinander trennen.

Fehlerintoleranz kann man auch als vorausschauende Fehlerbehandlung ansehen. Ihre Methoden sind Auswahl sehr zuverlässiger Bauelemente und Entwurf von sicheren Verbindungen. Bei der Softwareentwicklung heißt dies:

- * Korrektheit und Vollständigkeit der Spezifikation prüfen

- * testen und verifizieren der Implementierung.

Die Wahrscheinlichkeit, daß eine Komponente oder Verbindung ausfällt, charakterisiert die Zuverlässigkeit des Gesamtsystems. Die Verfügbarkeit ergibt sich aus dem Produkt der Verfügbarkeiten aller Einzelkomponenten.

Fehlertoleranz dagegen bietet die Möglichkeit, daß auch bei Ausfall von einzelnen Elementen weiter Aktionen durchgeführt werden können, wenn auch mit verminderter Leistung.

Das Maß für die Fehlertoleranz ist die Redundanz, der Anteil für zusätzliche Komponenten und Ausführungszeiten am Gesamtaufwand.

Es gibt drei Formen von Redundanz:

* Hardware-Redundanz

Bei den Bauteilen kann sie als schützende Redundanz beigelegt werden. Man unterscheidet zwischen maskierender (oder statischer) und dynamischer Redundanz.

Ein System mit maskierender Redundanz besteht aus mindestens drei gleichartigen Bauelementen und einem Vergleicher, der die Ergebnisse prüft und eine Mehrheitsentscheidung fällt.

Bei der dynamischen Redundanz wird der Fehler angezeigt und die fehlerhafte Komponente durch neue Bauteile ersetzt.

Kombinationen beider Arten bezeichnet man als hybride Redundanz.

* Software-Redundanz

Hierunter versteht man Softwarekomponenten, die eigens zur Fehlerbehandlung entstanden sind. Das können Programme sein, die mehrfach implementiert sind, zusätzliche Datenkomponenten zur Sicherung des Zustands, Diagnoseprogramme oder Fehlerkompensationsroutinen (error-handler).

* Zeitredundanz

Darunter fallen Wiederholungen bei fehlerhaften Abläufen (diese Methode ist vor allem bei transienten Störungen sinnvoll) und die erneute Durchführung von Aktionen nach einem Systemausfall.

1.6 Phasen der Fehlerbehandlung

Der Übergang von einem inkonsistenten Zustand in einen fehlerfreien, von dem aus man wieder weiterarbei-

ten kann, läßt sich in vier Phasen unterteilen:

1) Fehlererkennung (Detektion)

Hierunter fallen alle Aktivitäten, die festzustellen versuchen, ob ein Defekt vorliegt. Sie sollten möglichst umfassend sein und oft prüfen, dabei aber selbst keine Fehler machen und den normalen Betrieb nur wenig belasten.

2) Bestimmen des Schadensausmaßes und Diagnose

Es ist zu diagnostizieren, welche Komponenten nicht mehr funktionsfähig sind. Die zerstörten Datenstrukturen müssen ausfindig gemacht werden. Da man die Routinen zur Lokalisierung von Defekten erst dann braucht, wenn ein Fehler vorliegt, nimmt man dabei auch längere Laufzeiten in Kauf.

3) Erstellen eines funktionsfähigen Zustands der Hardware (Rekonfiguration)

Dieser Teil hat zu gewährleisten, daß in der weiteren Bearbeitung ein System aus Hardwarekomponenten ohne Defekte zur Verfügung steht. Die Aufgaben müssen eventuell neu verteilt werden. Die vorliegende Arbeit geht auf diesen Aspekt nicht näher ein.

4) Wiederherstellen eines konsistenten Zustandes (Recovery)

Die Daten sind so zu rekonstruieren, daß ein Zustand hergestellt wird, der konsistent ist und der Situation vor dem Auftreten des Fehlers am ehesten entspricht.

Nach dieser Phase sollte eine normale Verarbeitung wieder möglich sein.

2. Fehlererkennung

Ausgangspunkt für jede Fehlerbehandlung ist das Erkennen, ob ein inkorrektes Verhalten vorliegt oder nicht. Um den Schaden möglichst klein zu halten, sollte ein Fehler möglichst bald erkannt werden. Theoretisch müßte man nach der Definition von "Fehler" im letzten Abschnitt bloß das tatsächliche Ergebnis einer Aktion mit ihrem zuvor spezifizierten Verhalten vergleichen, um zu prüfen, ob ein Defekt vorliegt.

Schwierigkeiten verursacht dies allerdings, wenn die Interpretation der Spezifikation und das System nicht unabhängig voneinander zu realisieren sind oder die Vergleichseinheit nicht zuverlässig arbeitet. Dann kann ein Fehler auch in diesem Idealfall unentdeckt bleiben.

Das Hauptproblem aber ist, eine Spezifikation so aufzubereiten, daß sie zum Vergleich mit der Implementierung herangezogen werden kann. Sind Spezifikationen umgangssprachlich formuliert, nicht eindeutig oder unvollständig, so ist die Wahrscheinlichkeit hoch, daß auch beim Umsetzen in Prüfvorschriften Mißverständnisse auftreten.

Formale Spezifikationen, die zumindest teilweise auf Vollständigkeit und Widerspruchsfreiheit hin untersucht wurden, können hier die Situation entscheidend verbessern. Ihre interpretative Abarbeitung und die Umwandlung des Ergebnisses sind aber immer noch mühevollen Prozesse, die auch bei einer formalen Festlegung der Anforderungen meist nicht effektiv automatisierbar sind.

Die Kosten für diese Art von Fehlererkennung sind aber auch dann für normale Anwendungsfälle nicht erschwinglich.

Man wird also abhängig von der Struktur eines Systems Fehlermodelle aufstellen und anhand der Zuverlässigkeitsanforderungen geeignete Prüfstellen und -methoden auswählen.

Ein weiteres Kriterium bei der Festlegung der Fehlererkennung sind die Auswirkungen von Defekten auf das Gesamtsystem. Wenn die Anwendung nicht gut strukturiert ist, führen unentdeckte oder zu spät erkannte Fehler mit hoher Wahrscheinlichkeit zum Zusammenbruch des Systems.

Um zu wissen, wie Fehler aufzuspüren sind, müssen die Ursachen von Fehlern und deren Klassifizierung näher analysiert werden.

2.1 Fehlerquellen

Bei Fehlern gibt es eine Vielzahl von Unterscheidungsmöglichkeiten. In konkreten Anwendungsfällen wird man die relevanten Klassen herauslösen und so ein typisches Fehlerprofil erstellen. Die hier behandelten Fehlerstellen beziehen sich vor allem auf Defekte, deren Behandlung unter Mitwirkung der Software geschehen kann.

Diese Klasse läßt sich noch zergliedern:

- 1) in welchem Entwicklungsstadium der Fehler induziert wurde (/Tra80/).

Hier soll festgestellt werden, ob er beim Entwurf, bei der Implementierung entstanden ist, oder ob er durch Wartung oder Alterungserscheinungen hervorgerufen wurde. Fehler, die in frühen Stadien entstehen und spät entdeckt werden, haben meist die fatalsten Folgen. Analysen haben gezeigt, daß weit über die Hälfte der Fehler, die während oder nach dem Abnahmetest entdeckt wurden, schon im Entwurf gemacht werden (Abbildung 2.1). Deshalb ist in den frühen Phasen des Projektes besonders groß-

er Wert auf Überprüfungen von Vollständigkeit und Widerspruchsfreiheit zu legen.

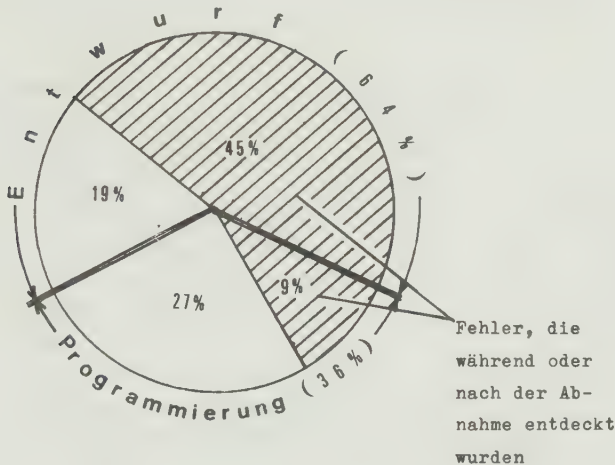


Abbildung 2.1:

Entstehung und Entdeckung von Fehlern, verteilt auf die verschiedenen Entwicklungsphasen (Quelle /Tra80/)

Fehler, die durch spätere Modifikationen in ein System eingepflanzt werden, können ebenfalls hohe Kosten verursachen. Wenn ein Defekt während der Betriebsphase behoben wird, so muß man mit einem zehn- bis hundertmal höheren Aufwand rechnen, als wenn er schon in der Anforderungsdefinition entdeckt wird (Abbildung 2.2). Da die Entwickler des Produktes zu dieser Zeit nicht mehr für dieses Projekt arbeiten, ist mit einer relativ langen Einarbeitungsphase zu rechnen.

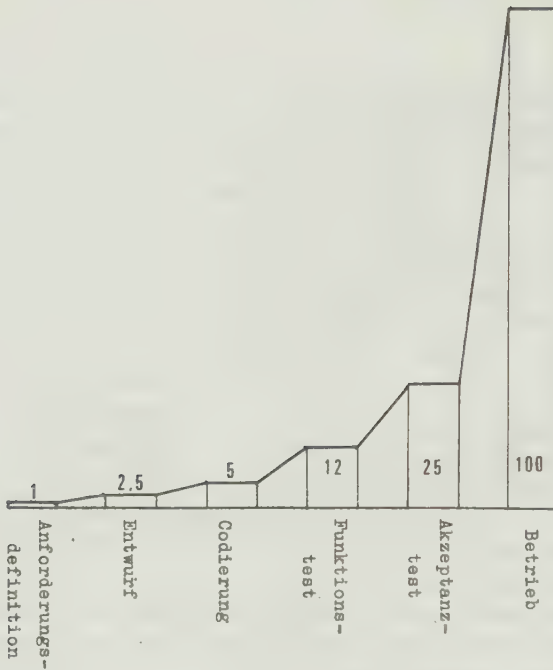


Abbildung 2.2:
Relative Kosten, um einen Fehler
zu finden (Quelle: /Tra80/)

2) zeitliche Dauer der Störung.

Hier wird vor allem unterschieden, ob es sich um einen permanenten Ausfall oder um eine vorübergehende (transiente) Störung handelt. Diese Unterscheidung spielt vor allem bei der Diagnose und Rekonfiguration eine große Rolle, weil bei tran-

sienten Fehlern diese Maßnahmen überflüssig sind. Die Unterscheidung ist deshalb auch von großer Bedeutung, weil es fatale Folgen hätte, wenn man wegen jeder kurzfristigen Störung gleich große Hardwarekomplexe auszutauschen würde.

3) determiniertes oder zufälliges Fehlerergebnis.

Fehler, die ein definiertes Resultat liefern, sind leichter zu identifizieren. Ein großer Teil von Hardwareausfällen (z.B. Kurzschluß, Leitungsbruch, Stromausfall) gehört dieser Klasse an. Treten nicht-deterministische Störungen auf, so erhöht sich der Aufwand für die Erkennung.

Leider beschäftigen sich viele Fehlererkennungsmethoden nur mit zufälligen Störungen. So geht man bei der Codierungstheorie oft von einem Modell aus, in dem alle Verfälschungen als gleichwahrscheinlich angenommen werden (siehe auch Abbildung 2.3).

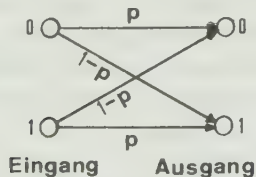


Abbildung 2.3:

"Symmetrischer Kanal"

(p , $1-p$: Übergangswahrscheinlichkeiten)

4) Verlauf nach dem Auftreten des Fehlers (/Jon79/).

Die beiden Extremfälle sind hier, daß defekte Komponenten überhaupt keine Aktionen mehr ausführen (passive Fehler), und daß die Aktivitäten mit unerwünschten Seiteneffekten begleitet scheinbar ord-

nungsgemäß weiterlaufen (aktive Fehler).

Ein Ausfall (crash) ist besser zu handhaben, da hier das Schadensausmaß leichter zu begrenzen ist. Fehlererkennung kann in größeren Abständen erfolgen. Bei aktiven Fehlern wird deshalb oft versucht, sie zu "passivieren", d.h. die fehlerbehaftete Einheit sperrt sich selbst oder wird, bevor ein Ergebnis nach außen geht, getestet.

5) Hardware- oder Softwaredefekte.

Dies ist ein häufig ausgetragener Streit zwischen den an einem Projekt beteiligten Gruppen. Das mag für die Anwendung selbst belanglos sein. Wenn man aber bedenkt, daß beide Komponenten oft von verschiedenen Vertragspartnern realisiert werden, so kann diese Frage zum Dreh- und Angelpunkt für alle Gewährleistungsansprüche und Schadensersatzforderungen werden.

Die Unterscheidung zwischen Hard- und Software ist vielleicht auch deshalb so kritisch, weil diese Themen von verschiedenen Fachleuten betreut werden. Ordnet man den Defekt falsch ein, so ist die Fehlersuche um einige Größenordnungen aufwendiger.

6) Defekte in Kommunikationswegen.

Fehler in einzelnen Elementen lassen sich durch Überprüfen der Schnittstellen leichter isolieren. Eine Verbindung, die zu einem falschen Ziel führt, kann folgende Komponenten stören:

- das Element, das eine Kommunikation initiiert
- den vermeintlichen Adressaten einer Nachricht
- die falsche Zielkomponente.

Aus diesem Grund sollen die Verbindungsstrukturen

eines Systems möglichst einfach und fehlersicher realisiert werden.

7) Beobachtbare oder versteckte Fehlerquellen.

Normalerweise wird man besser auf einen Defekt reagieren können, wenn dessen Ursachen offenliegen. Aber andererseits will man bei der Benutzung einer Komponente nicht zu sehr auf unwichtige Details eingehen.

Beobachtbar ist ein Defekt dann, wenn die ihn auslösende Operation dem Aufrufer genügend Information zur Verfügung stellt, mit der dieser sinnvoll eine Fehlerbehandlung durchführen kann. Durch Abstraktion gehen aber Informationen an den Schnittstellen zwischen den Modulen verloren, so daß auch mit nicht beobachtbaren Fehlern zu rechnen ist.

8) Ausdehnung von Fehlern.

Je mehr Komponenten von einem Defekt betroffen sind, desto schwieriger ist es, einen konsistenten Zustand wiederherzustellen. Dabei wird aber oft übersehen, daß nicht nur physikalische Abhängigkeiten die Ausbreitung fördern, sondern auch logische. Da aber solche Zusammenhänge nie ganz beseitigt werden können, sind die Nahtstellen (Singularitäten) der Komponenten besonders sorgfältig zu entfernen.

Manche Behandlungsmethoden eignen sich, wie später noch gezeigt wird, besonders gut für bestimmte Fehlertypen. Doch neben der Art des Fehlers bestimmen die Einsatzgebiete die Methodik der Fehlererkennung. So ist zu unterscheiden, wie lange das System unterbrechungsfrei in Betrieb sein muß (Missionszeit) und wie hoch die Zuverlässigkeitsanforderungen der Anwendung ist.

2.2 Prüfstellen

Oberstes Ziel jeder Fehlerbehandlung ist, einen Defekt möglichst rasch zu erkennen, um den daraus entstehenden Schaden klein zu halten. Da aber, wie oben erwähnt, die Erkennung eines Fehlers auch einen bestimmten Aufwand verursacht, sind Prüfstellen so zu optimieren, daß die Gesamtkosten für Fehlerbehandlung minimal werden.

Mögliche Prüfstellen sind:

1) vor Beginn einer Aktion

Dieses Prüfen minimisiert den Folgeschaden eines Fehlers, da noch keine Veränderungen vorgenommen wurden. Es ist aber für manche Aufrufe schwierig, allein anhand des aktuellen Zustands und der Eingabeparameter Fehler aufzudecken, ohne Veränderungen vorzunehmen. Wenn mehrere Operationen ein Objekt parallel modifizieren, müßten alle Aktivitäten miteinbezogen werden. Wenn es zu Interferenzen mit später gestarteten Aufrufen kommen kann, so müßte auch das einkalkuliert werden. Parallele Aktivitäten lassen sich auf diese Art und Weise also nur schlecht prüfen.

Transiente Fehler können ebenfalls nicht erkannt werden.

2) während einer Aktion

Die Fehlererkennung ist hier im allgemeinen dem Ort des Auftretens am nächsten. So kann besonders flexibel auf eine Vielzahl unterschiedlicher Fehler reagiert werden. Jedoch ist diese Behandlung im Detail sehr stark von der Struktur des Systems abhängig, so daß der Programmierer hier leicht die Übersicht verliert. Der Aufwand, jeden Fehler lokal zu behandeln, erhöht die Komplexität des Systems

gewaltig.

3) am Ende einer Aktion

Der Programmierer hat hier die größte Menge an Information, um einen Fehler zu erkennen. Jedoch ist der Aufwand für die Behebung der Fehler beträchtlich höher als zu Beginn der Aktion. Zwar können auch transiente Fehler erkannt werden, jedoch kann der Defekt schon in andere Komponenten eingeflossen sein.

4) nicht direkt bestimmten Aktionen zugeordnete Testroutinen

Dabei kann durch geeignete Einstellung des Systemoverhead für die Fehlerbehandlung sehr variabel gestaltet werden, ohne die Anwendungsprogramme manipulieren zu müssen. So wird ein System wechselnden Anforderungen bezüglich Zuverlässigkeit am besten gerecht. Wieweit aber dabei das Ziel, den Schaden einzudämmen, erreicht wird, läßt sich bei dieser Methode schwer vorhersagen.

2.3 Erkennungsmethoden

Das Feststellen von Fehlern sollte

- * einfach zu implementieren sein, um nicht dadurch zu viele Fehler zu induzieren
- * wenig Aufwand verursachen, um die Leistungsfähigkeit auch bei gesteigerter Zuverlässigkeit zu gewährleisten (Aufwand bezieht sich hier auf Zeitbedarf, Speicherplatzanforderung und Implementierungskosten).
- * möglichst allgemein sein, um eine große Zahl von Fehlern aufzuspüren.

Diese widersprüchlichen Ziele lassen sich wohl kaum mit einer einzigen Art der Fehlererkennung verwirklichen. Deshalb werden hier gleich mehrere Methoden vorgestellt, die nach den Erfordernissen der Anwendung einsetzbar sind.

Arten der Fehlererkennung (Checks):

1) Replikation

Darunter fallen vor allem Vervielfältigung wichtiger Informationen. Gegen das Fehlverhalten der Datenträger kann man sich schützen, indem man wichtige Informationen mehrfach abspeichert.

Eine andere Methode, die der Bauelementetechnik entstammt, ist das Vervielfältigen ganzer Systeme. Sie wird vor allem angewandt bei Anlagen, die unterbrechungsfreien Betrieb erfordern, wie Flugüberwachung und Telefonvermittlung.

Wiederholung von Abläufen ist eine dritte Möglichkeit, durch Replikation Fehler zu erkennen. Sie findet deshalb kaum Verbreitung, weil man sich dabei vor Komponentendefekten u.ä. nicht schützen kann.

Im allgemeinen ist diese Art weniger an der Fehlererkennung orientiert als an deren Auflösung. Entscheiden, wo ein Fehler steckt, kann man erst ab drei Instanzen.

Die Analyse, ob ein Defekt vorliegt, ist wenig komplex und erfordert nur einen Vergleich. Die Kosten für Speicherplatz oder Zeit dagegen erhöhen sich gewaltig. Ist aber ein Fehler entdeckt, so ist seine Behebung oft sehr einfach.

Entwurfsfehler können nicht erkannt werden, wenn man die Systeme einfach vervielfältigt.

2) Invertieren (reversal check)

Auf das Ergebnis einer Funktion f wird deren Inverse angewandt und das Resultat mit den Eingabeparametern verglichen. Wichtigste Voraussetzung dafür ist, daß die Umkehrung existiert und sich möglichst unabhängig von f verwirklichen läßt. Beide Bedingungen sind in der Praxis nur selten in akzeptabler Form erfüllt.

Wenn man nur die Forderung erhebt, daß durch Invertieren ein Bereich gefunden wird, in dem die Eingangsparameter liegen müssen, erleichtert sich vieles. Ein Beispiel für diese modifizierte Anwendung ist die Überprüfung von Optimierungsaufgaben, wo man feststellt, ob die Lösung überhaupt die Ausgangsbedingungen erfüllt.

Da man von einer anderen Situation ausgeht (vom Ende der Operation) ist die Wahrscheinlichkeit hoch, daß andere Operationen verwendet werden als bei der Durchführung der Aktion.

Die Schadensbehebung ist bei dieser Methode nur mit hohem Zusatzaufwand möglich. Es gibt keine genaue Analyse um festzustellen, wo der Fehler nun tatsächlich auftrat.

3) Codieren

Da eine Vervielfältigung der Information für die Erkennung von Defekten sehr aufwendig ist, wird bei der Speicherung und dem Transport von Daten häufig auf die Codierung zurückgegriffen.

Codieren setzt sich zusammen aus dem Hinzufügen von redundanten Informationen und dem Abbilden in einen anderen Zeichenvorrat. Man kann damit die Sicherungsinformation stark komprimieren und in relativ einfachen Rechenschritten die Gültigkeit der Daten prüfen. Diese Methode dient der Fehlererkennung und -behebung. Der Speicheraufwand

kann den Bedürfnissen nach Zuverlässigkeit angepaßt werden. Die typischen Beispiele dafür sind Paritätsbildung, Prüfsummen und Divisionsrest.

Die Vorzüge der Codierung will man nicht nur beim Datentransport und der Speicherung nutzen, sondern auch bei arithmetischen und logischen Operationen (/Pra80/). Dadurch könnte auf das Codieren/Decodieren bei jeder Funktion verzichtet werden. Einen Code für solche Zwecke zu finden, erscheint aber nach heutigem Stand nicht möglich.

Weiterentwicklungen bei Codes betrachten auch nicht gleichverteilte Fehlerwahrscheinlichkeiten. In Fällen, wo Fehler ein determiniertes Ergebnis liefern, können mit n -aus- m Codes alle Fehler erkannt werden.

4) Schnittstellenüberwachung

Ziel dieser Überwachung ist es, an den Grenzen zwischen Komponenten deren Interaktionen zu überwachen. Damit ist eine Ausbreitung von Fehlern über die einzelnen Elemente hinaus am wirkungsvollsten einzudämmen. Diese Methode findet sich auch im realen Leben häufig: bei der Überwachung von Gebäuden, dem Zufluß von Energie usw.

Im Fehlerfall braucht nur das defekte Modul ausgetauscht werden, wenn die Kontrollen erfolgreich waren.

Auf Programmebene gehören Zeitüberwachungen und Rechtsprüfungen zu den geläufigsten Überwachungen dieser Art.

5) Testprogramme

Durch Anwendung vorgegebener Operationen wird in definierten Zeitabständen oder aufgrund spezifischer Ereignisse überprüft, ob bestimmte Komponenten noch ordnungsgemäß funktionieren. Da jedoch

beim Start dieser Aktion noch nicht bekannt ist, welche Komponenten überhaupt noch richtig funktionieren, ist die Reihung dieser Tests schwierig. Jedes Prüfprogramm geht davon aus, daß bestimmte Teile richtig funktionieren.

Beispiel dafür ist der Selbsttest einer Zentraleinheit. Überprüft man zuerst den Prozessor, so können bei Bus- oder Speicherfehler durch den Test weiter Daten zerstört werden. Will man eine der anderen Komponenten zuerst untersuchen, so ist nicht gewährleistet, daß der Prozessor diese Routinen wie gewünscht durchführt.

Solche Diagnosen sollten deshalb besser von außen überwacht und durch einen zuverlässigen "Hardcore" ausgewertet werden.

6) Zeitüberwachung

Diese Art von Kontrolle erweist sich vor allem in Echtzeitsystemen als wirksam. In vielen Fällen ist dann kalkulierbar, in welchem Abstand ein Folgeereignis eintreffen muß oder wie groß die Reaktionszeit auf eine Ausgabe sein darf. Viele Aktivitäten müssen innerhalb eines bestimmten Zeitintervalls abgewickelt werden, damit die Anlage unter Kontrolle bleibt. Wenn die Zeitbedingungen klar identifiziert werden, können sie auch leicht überwacht werden und so einen wertvollen Beitrag zur Fehlererkennung liefern.

7) Plausibilitätskontrollen

Diese Kennzeichnung trifft auf eine große Anzahl von Erkennungsmethoden zu. Dabei werden leicht zu ermittelnde Bedingungen zu bestimmten Zeitpunkten überprüft. Im Gegensatz zu den Zusicherungen bei der Verifikation braucht die Plausibilitätskontrolle allerdings nicht vollständig zu sein.

8) Strukturprüfungen

Wenn große Datenmengen zu verwalten sind, so benötigt man Informationen, die den Zugriff darauf erleichtern. Diese zusätzlichen Daten können nun dazu herangezogen werden, die Konsistenz zu prüfen. Die Anwenderinformation kann auf diese Art und Weise natürlich nicht kontrolliert werden. Wenn aber Anwender- und Verwaltungsdaten zusammen abgespeichert werden, so ist die Wahrscheinlichkeit hoch, daß Defekte sich auch auf beide Datenbereiche ausdehnen.

So ist es bei indexsequentieller Speicherung von Daten unwahrscheinlich, daß ausgerechnet die Schlüssel übereinstimmen, wenn eine Datenseite an falscher Stelle abgespeichert wird.

Die meisten Methoden verzichten darauf, das gleichzeitige Auftreten von mehr als einem Fehler zu behandeln. Auch wird eine Fehlerquelle oft nicht zu jedem Zeitpunkt berücksichtigt (/Hop80/).

Unerwünschte Seiteneffekte können ebenfalls kaum analysiert werden. Wie Tests dieser Art die Komplexität erhöhen können, zeigen deutlich bitmustersensitive Speichertests. Der Aufwand steigt exponentiell mit der Zahl der Speicherplätze, während eine lineare Zahl von Schritten genügt, um Fehler des Typs "ständig-0" und "ständig-1" zu entdecken.

2.4 Diagnose

Wenn man erkannt hat, daß ein fehlerhafter Zustand vorliegt, so läßt sich im allgemeinen noch nicht sagen, was die Ursache dieses Defektes ist.

Es müssen Funktionen im System angestoßen werden, die schadhafte Stellen finden, um sie von den korrekten

Teilen isolieren zu können. Zu diesem Zweck zerlegt man das System in voneinander unabhängige Module und läßt sie gegenseitig testen. Welche Komponenten Aussagen über die Funktionstüchtigkeit der andern Elemente machen können, wird im Diagnosegraphen festgehalten. Die Knoten stellen die unabhängigen Teilsysteme dar. Eine gerichtete Kante von a nach b gibt an, daß a Komponente b testen kann (/Fri80/).

Das häufig zugrundeliegende Fehlermodell geht von folgenden Annahmen aus:

- 1) Das System besteht aus Komponenten, die unabhängig voneinander andere Elemente testen können.
- 2) Die Testergebnisse sind "funktionstüchtig" (r) oder "fehlerhaft" (f). Diese Resultate stellen eine Bewertung der Kanten dar, die in ihrer Gesamtheit "Syndrom" genannt werden.
- 3) Das Testergebnis ist richtig, wenn die testende Einheit fehlerfrei ist, ansonsten ist es undefiniert.
Alle möglichen Fehler werden also durch einen Test erkannt. Flüchtige Fehler sind dabei nicht miteinbezogen.

Abbildung 2.5 zeigt die möglichen Syndrome für den Diagnosegraphen aus 2.4 für den Fall, daß genau ein Defekt vorliegt bzw das System in fehlerfreiem Zustand ist.

Defekte in einer Menge von Einheiten N_i und N_j sind unterscheidbar, wenn die möglichen Syndrome bei N_i und N_j verschieden sind. Wenn es ein Syndrom gibt, daß sowohl bei Versagen der Komponenten N_i als auch N_j auftritt, so sind die Defekte N_i und N_j nicht unterscheidbar. Beispiele für nicht unterscheidbare Fehlermengen bei dem Diagnosegraphen in Abbildung 2.4 sind

$N_i = \{n_1, n_2\}$ und $N_j = \{n_1\}$.

Es gibt zwei Maße für Diagnostizierbarkeit, 1-Schritt- und t-Schritt-Diagnostizierbarkeit.

Im ersten Fall können alle fehlerhaften Einheiten in einem Diagnoseschritt ohne Rekonfiguration erkannt werden, wenn die Zahl der Defekte t nicht übersteigt.

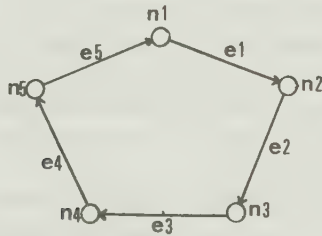


Abbildung 2.4:
Diagnosegraph

e1	e2	e3	e4	e5	fehlerhafte Komponente
r	r	r	r	r	-
r	r	r	r	f	n1
f	r	r	r	f	n1
f	r	r	r	r	n2
f	f	r	r	r	n2
r	f	r	r	r	n3
r	f	f	r	r	n3
r	r	f	r	r	n4
r	r	f	f	r	n4
r	r	r	f	r	n5
r	r	r	f	f	n5

Abbildung 2.5:

Mögliche Syndrome, wenn höchstens ein Fehler auftritt

Sequentielle Diagnose oder t -Schritt-Diagnose ist durchführbar, wenn in jedem Schritt mindestens eine fehlerhafte Einheit erkannt wird, unter der Voraussetzung, daß höchstens t Fehler vorhanden sind. Im schlimmsten Fall wären t Schritte erforderlich, wobei jeder Schritt aus Diagnose und anschließendem Entfernen der als fehlerhaft erkannten Elemente besteht.

Auf diesen Annahmen aufbauend wurden Graphen untersucht. Es lassen sich notwendige und hinreichende Bedingungen zur Diagnose der zugrunde liegenden Systeme angeben.

Das wichtigste Resultat für die 1-Schritt-Diagnostizierbarkeit ist, daß die Zahl der Knoten im System N größer als $2t$ sein muß und jede Einheit von mindestens t anderen getestet werden muß. t ist dabei die Höchstzahl von Fehlern, die durch die Diagnose sicher erkannt werden können.

Ziel der Untersuchungen mit diesem Modell ist es auch, Strukturen zu finden, die mit einem minimalen Aufwand eine gute Diagnose ermöglichen.

Die Einschränkungen, die durch dieses Modell impliziert werden, können in vielen Anwendungen nicht eingehalten werden. Mögliche Erweiterungen des Modells für realistische Anwendungen wären:

- 1) Das vollständige Testen erfordert mehr als eine testende Einheit. Jede Komponente erkennt nur bestimmte Fehler in anderen Bereichen.
- 2) Es werden auch Systeme behandelt, deren Einheiten sich abhängig von der Fehlerrate gegenseitig testen. Manchmal ist es vorteilhaft, die stochastische Natur von Fehlern besser auszunutzen. Wenn es mehrere Möglichkeiten zur Interpretation des Testergebnisses gibt, wird diejenige ausgewählt, die eine höhere Wahrscheinlichkeit besitzt.

- 3) Vollständiges Testen ist nicht möglich. Auch eine von fehlerfreien Moduln als korrekt angesehene Einheit kann defekt sein. Es können also auch Fehler unentdeckt bleiben. Diese Erweiterung erfordert allerdings dann, daß die gewonnenen Aussagen wahr-scheinlichkeitstheoretisch näher analysiert werden müssen.
- 4) Einbeziehen von flüchtigen Fehlern. Auch unter diesen Voraussetzungen ist keine vollständige Diagnose möglich.
- 5) Diagnose während des laufenden Betriebs. Dabei sind Teststrategien zu entwerfen, die zu jeder Zeit nur einen Teil der Module in Anspruch nehmen. Da meistens Reservemodule vorhanden sind oder manche Aufgaben nicht die gesamte Konfiguration erfordern, bietet es sich an, die zur Zeit nicht aktiven Elemente zu einem Subgraphen zusammenzufassen und diesen zu diagnostizieren.

In manchen Anwendungsfällen ist es gar nicht notwendig, die Zahl der defekten Module exakt zu ermitteln. Es genügt, daß eine Obermenge der defekten Module identifiziert wird. Damit wird für manche Anwendungsfälle eine effiziente Diagnose erst möglich. Mit dem Beispiel in Abbildung 2.4 und 2.5 läßt sich diese Behauptung belegen. Man kann auch zwei fehlerhafte Komponenten entdecken, wobei es aber nicht möglich ist, bei genau einem Defekt festzustellen, ob auch das nachfolgende Element Mängel aufweist.

Bei der Lösung von realen Problemen hat sich das hierarchische Diagnosemodell bewährt (/Neu80/). Man geht von einer Menge intakter Komponenten als "Hardcore" H aus. Mit H testet man weitere Bestandteile des Systems. Bei positivem Ausgang übernimmt man diese Kom-

ponenten in H. Dieser Vorgang wird solange fortgesetzt, bis H alle Elemente des Systems enthält. In diesem Fall liefert die Diagnose die Anzeige "fehlerfrei". Ansonsten wird angezeigt, bei welcher Komponente der Test abgebrochen wurde (Abb. 2.6).

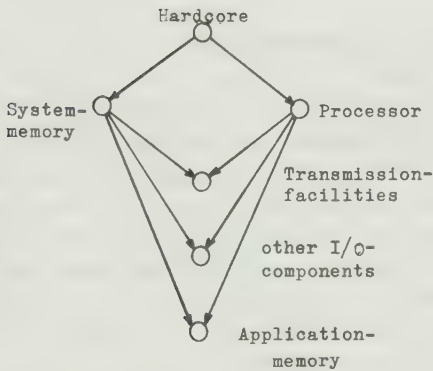


Abbildung 2.6:
Hierarchische Diagnose

3. Fehlermaskierung und -eindämmung

Nach der Darstellung von Fehlererkennung und -lokalisierung soll nun erläutert werden, wie der Schaden möglichst klein zu halten ist (auf Software-/Hardwareebene) und wie man defekte Komponenten aus dem System entfernen kann.

3.1 Fehlerbeseitigung auf Hardwareebene

Auf Maßnahmen zur Behandlung von Fehlern per Hardware sei hier nur kurz eingegangen, diese werden detaillierter an anderen Stellen (z.B. /Mae77/) dargestellt. Es geht dabei im wesentlichen darum, defekte Module durch funktionierende zu ersetzen oder fehlerhafte Resultate durch Vergleich und Mehrheitsentscheidung nicht wirksam werden zu lassen. Dabei muß das Umschalten oder Auswählen des richtigen Ergebnisses mit hoher Wahrscheinlichkeit korrekt erfolgen.

Als Mittel zur Beseitigung defekter Komponenten setzt man Redundanz ein. Sie kann auf drei verschiedene Arten realisiert werden:

1) statische Redundanz

Dabei befinden sich alle Module in vollem Betrieb. Die Eingänge sind mit allen Elementen verbunden. Die Ausgänge werden in einem Vergleicher (Voter) geprüft, und wenn mindestens m Elemente das gleiche Ergebnis liefern, wird dieses weitergeleitet (m -aus- n -Entscheidung).

Die einfachste und bekannteste Version von statischer Redundanz ist die "Triple-Modular-Redundancy" (TMR), wobei das System aus drei gleichartigen Elementen besteht und mindestens zwei die gleichen

Ausgangssignale erzeugen müssen, um ein korrektes Ergebnis zu erhalten. Tritt also in einem der Module ein Fehler auf, so wird er vom Vergleicher "maskiert". Daher wird diese Methode auch maskierende Redundanz genannt.

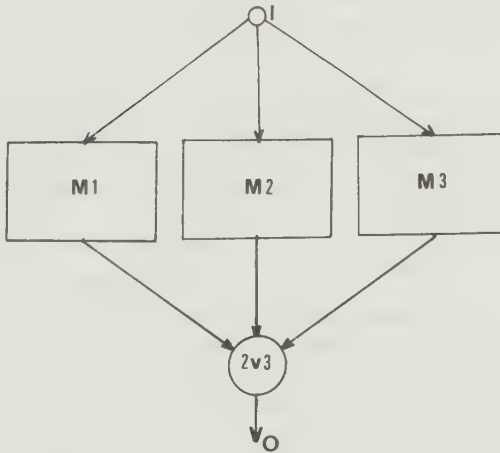


Abbildung 3.1:
Triple-Modular-Redundancy

Vorteile bieten der relativ einfache Aufbau und die problemlose Beseitigung transienter Fehler. Erkennung und Behebung von Defekten sind miteinander verknüpft. Als kritisch für die Gesamtverfügbarkeit erweist sich der Vergleich. Da er jedoch meist sehr einfach aufgebaut werden kann und es erprobte Verfahren zum Maskieren von Fehlern gibt, dürfte das erfahrene Entwickler nicht vor unlösbare Probleme stellen.

Ein gravierender Nachteil ist, daß kein automatisches Entfernen defekter Module vorgesehen ist.

Für lange Betriebszeiten ohne Wartung wird die Wahrscheinlichkeit, daß ein System diese Dauer in fehlerfreiem Zustand überlebt, durch Einsatz der TMR geringer (/Mae77/). Ohne periodische Überprüfungen kommt man also auch bei der "N-Modular-Redundancy" (NMR) auf lange Sicht nicht aus.

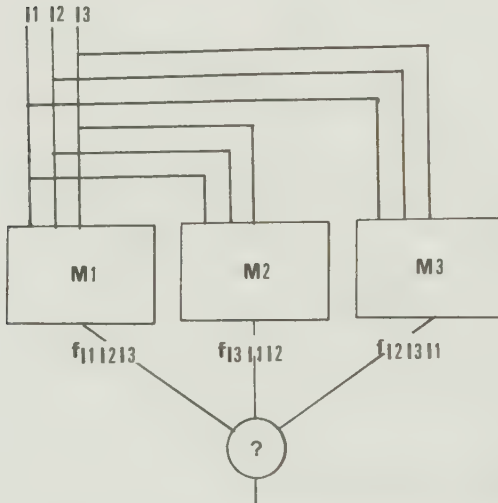


Abbildung 3.2:

Mögliche Ergebnisse bei sequentieller Bearbeitung gleichzeitig eintreffender Ereignisse

Ein weiteres Problem bei dieser Methode stellt die Synchronisation des Ablaufs zwischen den Modulen dar. Je komplexer der innere Aufbau ist, desto mehr variieren die Bearbeitungszeiten und damit oft auch die Bewertung der Eingangsdaten sowie die davon abhängigen Resultate. Dies liegt darin begründet, daß trotz gleichen Aufbaus die Bauelemente Schwan-

kungen unterliegen, die zu einem anderen Zeitverhalten der Systeme führen. Wenn externe Ereignisse auf unterschiedliche interne Zustände treffen, so ist oft die Reihenfolge der Ausgabe oder gar das Ergebnis selbst verschieden. Dies ist besonders kritisch, wenn Analogeingaben mit berücksichtigt werden. Ein konventioneller, einfach aufgebauter Voter hat Probleme, diesen Ausgabestrom zu interpretieren.

Dies kann dazu führen, daß Computer, auf diese Weise gekoppelt, "aus dem Tritt" kommen. Bei zentral getakteten Systemen lassen sich diese Gefahren reduzieren. Ganz aber wird man dieses Problem nie beseitigen können.

2) Dynamische Redundanz

Die Nachteile der NMR-Technik lassen sich teilweise bei der dynamischen Hardwareredundanz vermeiden. Wird hier eine Komponente defekt, so ersetzt man sie durch ein bisher inaktives Reservemodul (Spare). Darum heißt diese Methode auch "Stand-by-Sparing".

Fehlererkennung kann dabei auf verschiedene Art und Weise durchgeführt werden (siehe dazu auch Kapitel 2). Wird ein Defekt entdeckt, so wird auf das Reservemodul umgeschaltet. Der Komponentenwechsel muß mit hoher Zuverlässigkeit erfolgen. Ebenso wie beim Vergleich bei der maskierenden Redundanz läßt sich das durch Maßnahmen auf tieferer Hardwareebene bewerkstelligen. Mit Stand-by-Sparing erreicht man hohe Zuverlässigkeit auch bei langen Missionszeiten.

Ein zentrales Problem beim Einsatz dieser Technik ist, an welcher Stelle eine defekte Komponente abgeklemmt und wie das "Spare"-Modul zugeschaltet wird. Eine Lösungsmöglichkeit besteht darin, den

Bus, der die Ausgangsleitungen der einzelnen Module verbindet, umzuschalten. Dadurch erspart man sich Hochlaufzeiten des Reservebausteins. Dann sind auch die "Spares" ständig in Betrieb, benötigen Energie und unterliegen den üblichen Verschleißerscheinungen. Wenn man aber nur die aktiven Komponenten mit Strom versorgt und im Störfall nur die Energiequelle dem Reservemodul zuschaltet, läßt sich dieses Problem umgehen. Die Zuverlässigkeit erhöht sich, da Bausteine in denen kein Strom fließt, eine längere Lebensdauer haben. Die dann notwendige Anlaufphase nach einem Fehler muß die Anlage allerdings verkraften können.

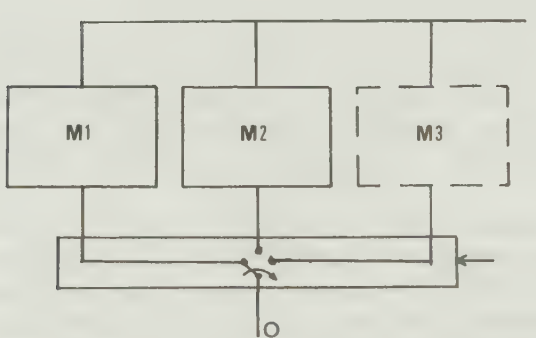


Abbildung 3.3:
Dynamische Redundanz

Nicht so ohne weiteres anwendbar ist diese Technik, wenn das aktive Modul Statusinformationen gesammelt hat, die beim Ausfall verloren gehen. Dann sind noch zusätzlich wichtige Ereignisse so zu archivieren, daß nach dem Umschalten die bisher passive

Komponente ohne zu großen Informationsverlust fortfahren kann.

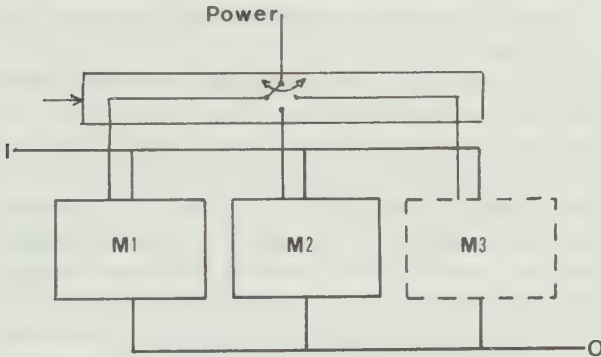


Abbildung 3.4:
Dynamische Redundanz mit umschaltbarer
Energieversorgung

Ein möglicher Vorteil der dynamischen Redundanz ist, daß die Spares auch zur Hintergrundverarbeitung herangezogen werden können (funktionsbeteiligte Redundanz). Nach einem Defekt kann man Aufgaben ganz fallengelassen. Das Gesamtsystem arbeitet mit verminderter Leistungsfähigkeit weiter (graceful degradation).

Die dynamische Redundanztechnik benutzt in Abwandlungen TANDEM für ihr NON-STOP-SYSTEM (TM). Dabei wird ein "Back-up"-Prozeß vom "Primary" in definierten Abständen mit Statusinformation versorgt. Im Fehlerfall übernimmt der "Back-up"-Prozeß beim "Checkpoint", dem als letztes übergebenen Status (/Tan77/). Diese Art der Bearbeitung tritt bei den neueren Produkten wie der "Transaction-Management-Facility" etwas in den Hintergrund.

3) hybride Redundanz

Die Vorzüge der statischen Redundanz (einfache Fehlererkennung) werden hier mit der weit höheren Verfügbarkeit des "Stand-by-Sparing" gekoppelt. Dabei wird das Ergebnis des "Voters" weiter ausgewertet und die bei der Abstimmung unterlegenen Module aus dem System entfernt.

Die kritischen Fragen zu dem Einsatz von Hardware-redundanztechniken für komplette Rechenanlagen bleiben dabei allerdings offen. Auch bei der hybriden Redundanz treten die Synchronisationsprobleme auf und die Notwendigkeit zum Sammeln von Statusinformationen bleibt - in einer abgeschwächten Form, wenn der Zustand der aktiven Module zugänglich ist. Diese Lösung ist allerdings extrem aufwendig (mindestens vier Komponenten plus Voter und Umschalt-einheiten), so daß sie nur in speziellen Anwendungsfällen einsetzbar ist.

3.2 Fehlereindämmung zum Schutz von Softwarekomponenten

Auch gegen Fehler von Programmen versucht man sich zu schützen. Es können dabei allerdings nicht ohne weiteres die Methoden eingesetzt werden, die auch für die Hardware Verwendung finden. Doch trotzdem lassen sich Parallelen bei der Erkennung und Behandlung von Defekten ziehen. Auch in der Software kennt man Teile des Systems, die bei der Fehlerbehandlung als korrekt vorausgesetzt werden. Man bezeichnet sie als "Kern" (Core). Auf der Bauelementeebene sind dies die Vergleicher und Umschalter, für die Programme bilden einige zentrale Teile des Betriebssystems die Kernfunktion.

Ein Grundprinzip, das jede Art von Fehlerbehandlung kennt, ist die Begrenzung der Auswirkungen, die ein Fehler haben kann. Dabei ist es notwendig, Defekte

möglichst bald zu erkennen.

Hier zeigt sich bereits ein gravierender Unterschied zwischen physikalischen und abstrakten Systemen. Während die Naturgesetze die Nebenwirkungen von bestimmten realen Aktionen stark einschränken, kann ein Programmierfehler fast beliebige Seiteneffekte erzeugen, wenn man sich nicht davor schützt. Ohne Sicherung kann man in jeder Anweisung auf fast alle Speicherbereiche, Register und E/A-Schnittstellen zugreifen und damit bei jeder noch so zuverlässigen Teilaufgabe Fehler injizieren. Deshalb ist es notwendig, diese Freiheitsgrade einzuschränken.

Mit der Idee der Modularisierung tauchte auch der Gedanke auf, den Arbeitsbereich eines Subjektes auf ein vernünftiges Maß zu reduzieren. Das Prinzip von den "kleinsten Privilegien" (/Den76/) kann zwar Fehler nicht verhindern, aber ihre Auswirkungen beträchtlich einschränken. Dadurch läßt sich leichter ermitteln, wer den Defekt verursacht hat, und die dafür verantwortliche Anweisung ist leichter und schneller zu lokalisieren. Damit allein ist die Fehlerbehandlung nicht abgeschlossen. Nur durch zusätzliche Bereinigungsmaßnahmen kann erreicht werden, daß das System danach wieder in vollem Umfang funktionsfähig wird.

Es ist nicht sinnvoll, beliebig kleine Bereiche zu schützen und die Domäne (=Menge der Objekte, auf die mit bestimmten Operationen aktuell zugegriffen werden darf) zu häufig zu wechseln, da sonst ein großer Teil der Bearbeitungszeit vom Schutzmechanismus beansprucht wird. Ein guter Kompromiß zwischen Leistungsfähigkeit und Sicherheit ist, Zugriffsrechte zumindest für die Dauer einer Operation auf einem Objekt zu vergeben. Ein Domänenwechsel ist dann nur notwendig, wenn ein neues Objekt zugegriffen wird. Wenn die Module so entworfen wurden, daß die Bindungen der Komponenten innerhalb des Objektes größer sind als zu den anderen Ele-

menten des Systems, lassen sich auch die Schutzaspekte effizient realisieren.

Die Schutzmechanismen in Software zu implementieren, ist nur mit großem Zeitaufwand möglich. Laufzeiten der Programme würden sich vervielfachen. Zu einer vernünftigen Implementierung kommt man, wenn in der Hardware mindestens Register vorhanden sind, die festlegen können, auf welche Speicherbereiche im Augenblick wie zugegriffen werden kann. Diese Register dürfen nur von dem Systemkern belegt werden. Dazu benötigt man eine spezielle Aktivierung des Systemkerns ("Supervisorcall" - SVC). Betrachtet man nur Fehler, die durch unbeabsichtigte Zugriffe entstehen, so kann der Schutzmechanismus einfacher aufgebaut werden als wenn man ein gezieltes Umgehen verhindern will.

Die besten Schutzvorrichtungen nutzen wenig, wenn sie durch Hardwaredefekte außer Kraft zu setzen sind. So kann zwar ein Zugriff auf einen zulässigen Bereich die Hardwarekontrollinstanzen passieren, aber durch einen Defekt in der Adressierlogik fehlgeleitet werden. Deshalb sollte man bei der Rückantwort prüfen, ob auch das richtige Element angesprochen wurde.

Damit lassen sich folgende Forderungen an einen Schutzmechanismus stellen:

- 1) Die Zugriffe müssen per Hardware überprüft werden (welches Segment, welche Zugriffsart und welche relative Adresse innerhalb des Segmentes)
- 2) Die korrekte Durchführung des Zugriffs muß überwacht werden
- 3) Ein Betriebssystemkern richtet Schutzdomänen ein und verändert sie. Diese Operationen sind nur in einem privilegierten Modus durchführbar und werden über spezielle Aufrufe aktiviert.

- 4) Es gibt systemeindeutige Identifikatoren (Id) für Objekte. Eine Modifikation dieser Id außerhalb des Systemkerns ist entweder unmöglich oder liefert mit einer hohen Wahrscheinlichkeit eine unzulässige Id (die zweite Alternative ist nur dann sinnvoll, wenn man ein gezieltes Umgehen der Schutzmechanismen nicht zu berücksichtigen braucht).

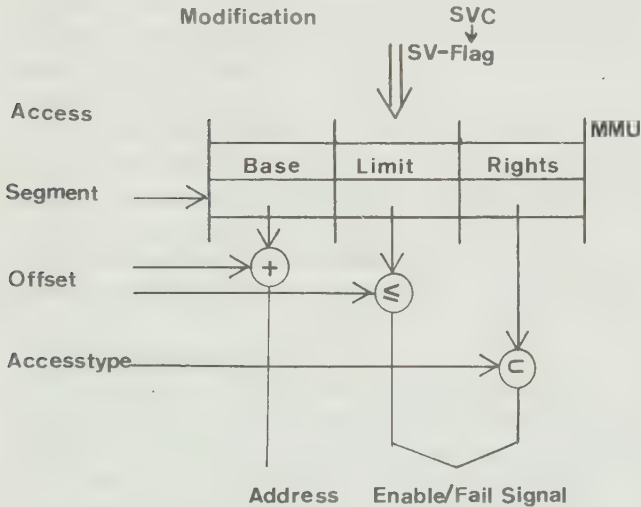


Abbildung 3.5:
Prinzipieller Aufbau der Hardware-
Speicherschuttkomponente

Mit Hilfe der Zugriffskontrollen und Fehlerisolation werden die Voraussetzungen für die Realisierung zuverlässiger Softwaresysteme geschaffen. Detailliertere Informationen kann man der einschlägigen Literatur entnehmen. Aber damit allein sind nicht alle Probleme gelöst. Dazu müssen noch weitere Methoden und

Werkzeuge geschaffen werden, aber auch ein entsprechendes Problembewußtsein bei den Programmierern. Wenn die Softwareentwickler kein Verständnis für die Problematik der Zuverlässigkeit aufbringen, wird die Anwendung kaum über eine längere Zeit in intaktem Zustand zu halten sein oder der Aufwand für die Sicherung der Statusinformationen wird immens hoch werden.

Mit diesen Problemen beschäftigen sich eingehend die nächsten Kapitel.

4. Fehlerbehandlung im Software-Entwicklungsprozeß

Wenn man Verfahren zur Fehlerbehandlung in Softwaresystemen untersucht oder entwickelt, dann taucht sofort die Frage auf, was diese Produkte von ähnlichen technischen Erzeugnissen unterscheidet, und was sie gemeinsam haben. Diese Analyse der grundsätzlichen Merkmale beeinflußt die Suche nach Methoden und Werkzeugen zum Aufdecken von Fehlerquellen und zur Recovery.

Je nachdem, unter welchem Gesichtspunkt man die Phasen der Entwicklung und des Betriebs betrachtet, wird man zu verschiedenen Aussagen kommen. Bei den folgenden Punkten zeigen sich wohl am deutlichsten die Gemeinsamkeiten und Unterschiede zwischen der "Software-technologie" und den anderen Ingenieurdisziplinen.

1) Entwicklungsmethoden

Hard- und Softwarekomponenten durchlaufen ähnliche Entwicklungsphasen - von der Anforderungsdefinition über den Entwurf und die Implementierung bis hin zu Installation und Wartung. Zwar sind die Gewichte bei den verschiedenen Anwendungen oftmals anders verteilt, jedoch läßt sich das weniger auf das Fachgebiet als auf die Aufgabenstellung zurückführen. Nur die Wartung von Programmen gestaltet sich meist anders als die der Hardware.

Die Entwicklung von Systemen vergleichbarer Größe organisiert man auf ähnliche Weise. Alle umfangreichen Projekte werden mit Teams wechselnder Größe durchgeführt.

Eine weitere Gemeinsamkeit der komplexen Anwendungen aus allen Bereichen ist die allgemeine Lösungs-

strategie. Man versucht das Problem durch Zerlegen in Teilaufgaben besser in den Griff zu bekommen. Diese "teile-und-beherrsche"-Strategie wiederum ermöglicht es erst, einen größeren Personenkreis an einem Projekt zu beteiligen.

2) Lösungsvielfalt

Bei den meisten Fachgebieten ist die Auswahl an Realisierungsmöglichkeiten der Problemstellungen gering. Durch die physikalischen Randbedingungen, Normen und den hohen Anteil an erprobten Verfahren in diesen Disziplinen gibt es oft nur wenig Spielraum für den erfahrenen Entwickler. Zwar sind auch hier im Rahmen der Projekte Neuschöpfungen notwendig, jedoch wird man sich aus Kostengründen weitgehend auf Bewährtes zurückziehen.

Anders ist die Situation bei der Programmentwicklung. Hier bieten sich zwar auch für jede Aufgabe eine große Zahl von Methoden und Werkzeugen an, die aber für die spezifische Problemstellung doch nur teilweise den Erfordernissen genügen. Diese Tendenz läßt sich bei einer großen Zahl von Softwareprodukten feststellen, wie Betriebssystemen, Compilern, Datenübertragungsprotokollen, Dateiverwaltungssystemen, aber auch bei Spezifikations- und Dokumentationspraktiken. Aus diesem Grund greifen die Projektleiter auch gern zu individuellen Lösungen. Die Nachfolger haben dann noch eine weitere Auswahlmöglichkeit an nicht kompatiblen Produkten.

Aber dieser Unterschied zu manch anderen Disziplinen ist nicht allein auf die rasche Fortentwicklung in der Informatik zurückzuführen oder der geringen Erfahrung auf diesem Neuland der Wissenschaft anzulasten. Es spielt auch die Architektur eine Rolle, die es mit relativ einfachen Mitteln erlaubt,

eine Vielfalt an leistungsfähigen Operationen durchzuführen. Diese Komplexität zu meistern, scheint wohl das Hauptproblem dieser Entwicklung zu sein.

3) Betrieb

Da Software an sich ein immaterielles Produkt ist, gibt es keine Alterungserscheinungen in herkömmlichen Sinne. Alle Fehler sind bereits bei der Inbetriebnahme eingepflanzt oder werden durch Änderungen nachträglich injiziert. Während Hardware-Wartung sich meist auf den Tausch von defekten Modulen beschränkt, befaßt sich die Software-Pflege mit Verbesserungen von Entwurfsmängeln oder mit der Beseitigung von nachträglich eingepflanzten Fehlern. Da fertige Programme relativ häufig modifiziert werden, ist die Wahrscheinlichkeit hoch, daß dabei neue Defekte entstehen. Software-Wartung ist deshalb keineswegs weniger aufwendig, erfordert aber ein weitaus qualifizierteres Personal als dies bei der Betreuung von Hardwarekomponenten notwendig ist.

Es ist jedoch trügerisch, in der Software das immaterielle Produkt zu sehen, bei dem man nur die Vielfalt der Lösungsmöglichkeiten zu beherrschen braucht und großen Wert auf Änderungsfreundlichkeit legen muß. Bei der Fehlerbehandlung ist auch die Umgebung mit einzuplanen. Technische Mängel in der Gesamtanlage oder fehlerhafte Bedienungen haben großen Anteil an den Störungsursachen. Der Aufbau der Programme und ihr Umfeld sind also maßgebend für die Zuverlässigkeit des Systems. Wie und in welchem Umfang dann die verschiedenen Störquellen einwirken, hängt dann noch weitgehend von der Art des Produktes ab.

Bei Compilern beispielsweise wird der Faktor "Umgebung" so wirksam, daß der Benutzer verlangt, möglichst

präzise zu erfahren, wo und wie er gegen die Sprachdefinition verstoßen hat. Außerdem erwartet man von einem Compiler, daß er syntaktisch korrekte Programme so übersetzt, daß sie auch ablaufen, wie in der Sprachdefinition angegeben ist. Da der erzeugte Objektcode und andere Listen jederzeit aus dem Quellprogramm wiedergewonnen werden können, ist es nicht nötig, diese Daten besonders zu schützen. Nur zur Sicherung vor unerlaubter Verwendung können Vorkehrungen notwendig werden.

Anders verhält es sich bei Datenbanken oder Datenverwaltungssystemen. Auch hier müssen sich die Aufrufe so verhalten, wie es in der Beschreibung angegeben ist. Darüber hinaus muß genau festgehalten sein, wie sich die fehlerhaften Aufrufe auf die Datenbasis auswirken. Die Konsistenz und Vollständigkeit der Daten muß trotz Fehlbedienungen, Systemzusammenbrüchen, Programmierfehlern und Hardwaremängeln mit einer großen Wahrscheinlichkeit gewährleistet sein, da der Informationsgehalt meist nur mit hohem Aufwand wiedergewonnen werden kann.

Bei Kommunikationsfehlern wird verlangt, daß trotz Übertragungsstörungen der Datenaustausch möglichst reibungslos verläuft. Bei Ausfällen von Komponenten dürfen Nachrichten nicht doppelt gesendet werden oder verlorengehen. Die Reihenfolge bestimmter Botschaften muß erhalten bleiben. Da es keine Instanz gibt, die über den Zustand aller Komponenten genau informiert ist, fällt es schwer, das Gesamtsystem konsistent zu halten. So kann es auch nicht verwundern, daß zwei Drittel des Codes für Netzknoten sich mit Fehlerbehandlung im weiteren Sinne beschäftigt.

Ein anderer wichtiger Anwendungsaspekt, der besondere Fehlerbehandlungstechniken erfordert, ist der Mensch-Maschine-Dialog. Hier ist die Bedienereingabe die große Fehlerquelle. Da Beziehungen zwischen den einge-

gegebenen Daten untereinander und im Verhältnis zu der Datenbasis sehr komplex sein können, ist es schwierig, die Gültigkeit einzelner Eingaben zu prüfen. Noch mehr Probleme bereitet es, bestimmte fehlerhafte Teilvorgänge rückgängig zu machen oder bei nachträglichen Korrekturen des Benutzers deren Auswirkungen nachzuvollziehen. Es genügt nicht, daß der Bediener eine bestimmte Position noch einmal anwählen kann und die Eingabe modifiziert. Alle Handlungen, die aufgrund der fehlerhaften Bedienung vorgenommen wurden, müssen rückgängig gemacht und dann mit den neuen Werten eventuell wieder angestoßen werden.

An diesen Beispielen sieht man sehr deutlich, wie Fehlerbehandlung den Anforderungen der Einsatzgebiete unterworfen ist.

4.1 Software-Entwicklungsprozeß

Jedes zuverlässige System stützt sich auf Komponenten, deren Versagen sehr unwahrscheinlich ist. Selbst bei einer hinreichenden Unterstützung durch erprobte Verfahrensweisen ist ein kontrolliertes Vorgehen während des gesamten Entwicklungsprozesses - die Wartung eingeschlossen - sicherzustellen. Deshalb muß die Softwaretechnologie als Methodenlehre des Entwicklungsprozesses Ausgangspunkt für weitere Überlegungen sein. Leider gibt es auf diesem Gebiet keine anerkannten Standards. Es existieren sogar verschiedene Anschauungen darüber, wie der "Lebenszyklus" der Software in Phasen zu unterteilen ist.

Einig ist man sich über das Ziel: Software soll leicht zu pflegen und zuverlässig sein. Dies wird am besten durch die klare Formulierung und Überprüfung eines jeden Schrittes erreicht. Dabei läßt sich feststellen, ob die getroffenen Maßnahmen auch richtig waren und in das Gesamtkonzept passen. Das Überwachen des Ent-

wicklungsprozesses soll dabei so gut es geht automatisiert werden, da eine manuelle Überprüfung einen großen Aufwand erfordert und meistens doch unvollständig bleibt.

Wie das im einzelnen zu erreichen ist, soll nun kurz an den Entwurfsphasen Anforderungsdefinition, Design, Implementierung, Inbetriebnahme, Pflege erläutert werden.

1) Anforderungsdefinition

Es geht darum, die Vorstellungen über das neue Produkt bzw. die Erweiterung an bestehenden Systemen möglichst vollständig und widerspruchsfrei zu formulieren. Dabei können nicht nur Rechnerspezialisten beteiligt sein. Auch Leute aus Fachabteilungen, die später die Anlage betreiben, haben bei der Festlegung der Anforderungen Mitspracherecht. Die beteiligten Leute unterscheiden sich in ihren Interessen und Anschauungen oft stark. Diese Differenzen aufzuheben, ist wohl eines der größten Probleme in dieser Phase. Trotz der Vielschichtigkeit sollte die Anforderungsdefinition möglichst formal sein, damit bei den späteren Entwicklungsstufen klar ist, was das Endprodukt leisten soll. Eine Sprache, die nur aussagt, was entwickelt werden soll und nicht wie, muß mit hinreichender Abstraktionsfähigkeit ausgestattet sein. Die resultierenden Anforderungsdefinitionen sollten mit Analyse- und Simulationsmöglichkeiten versehen werden können. Damit können unvollständige oder redundante Teile (/Ram81/) erkannt werden. Aber es erscheint kaum möglich, daß man Werkzeuge anbieten kann, mit denen jede Fachrichtung genau das einfach ausdrücken kann, was für ihre Anwendungen relevant ist. So stellt man rasch fest, daß jedes zur Zeit angebotene Hilfsmittel auf diesem Gebiet manche Anwendungsbereiche

besser und manche schlechter unterstützt. Es braucht aber nicht unbedingt eine Schwäche zu sein, wenn die Werkzeuge anwendungsorientiert sind, solange vergleichbare Sachverhalte auch in den verschiedenen Sprachen ähnlich dargestellt werden.

Existierende Produkte für die Darstellung der Anforderungen sind Requirements Statement Language (RSL), Problem Statement Language (PSL) und Structure Aided Design Technique (SADT).

Requirements Statement Language(RSL) (/Bel77/ und /Alf77/) erfüllt am ehesten die Bedingungen, die an eine Sprache für die Anforderungsdefinition gestellt werden. Sie ist jedoch relativ umfangreich und deshalb nicht leicht zu verstehen. Sie wird daher auch eher als Spezifikationssprache eingesetzt, als zum Erfassen der Anforderungen. Es gibt zu RSL ein Paket von Programmen (REVS), mit denen aktive RSL-Elemente (ALPHA's) mit den passiven Komponenten (DATA's) und deren Verknüpfungen (R-NET's) analysiert werden können.

Mit Hilfe von PSA (Problem Statement Analysis) lassen sich in PSL (Problem Statement Language) formulierte "Prozesse" auswerten und deren Interaktionen aufzeigen (/Tei77/). Mit diesem Produkt der Universität Michigan hat man aber ähnliche Probleme wie mit RSL. Der Aufwand, ein Problem vollständig mit den zur Verfügung stehenden Mitteln zu beschreiben, ist auch hier sehr hoch.

Bei der Structure Aided Design Technique (SADT) wird versucht, mit graphischen Darstellungen die Anforderungsdefinition durchschaubarer zu machen (/Ros77/). Jedoch gibt es bei dieser Methode nur wenige Hilfsmittel, mit denen man das resultierende Produkt analysieren könnte. Wie der Name schon sagt, ist dieses Werkzeug auch für den Detailentwurf einsetzbar.

Keine dieser Methoden hat bisher weite Verbreitung gefunden. Die graphische Form bietet den Vorteil, sehr knapp und verständlich Zusammenhänge darstellen zu können. Das verbessert die Kommunikation unter den Fachleuten erheblich. Doch diese Art der Darstellung ist nicht optimal von den herkömmlichen Rechnerstrukturen aus zu verarbeiten.

Manche Teilprobleme lassen sich besser durch Textdarstellung formulieren. Eine sinnvolle Kombination beider Darstellungsformen kann die besten Ergebnisse liefern.

Wie die Anforderungsdefinition aussehen sollte, hängt jedoch von vielen Faktoren ab. Nicht nur der Stand der Technik und die Verfügbarkeit der Werkzeuge spielen dabei eine Rolle, sondern auch der Ausbildungsstand der Beteiligten, Projektorganisation und andere soziologische, psychologische und wirtschaftliche Gesichtspunkte.

2) Design

Hier kommt es auf die genaue Formulierung an, wie die Komponenten des Produktes zusammenspielen sollen. Die prinzipielle Methode dieses Entwicklungsschrittes ist die Zerlegung des Problems in einzelne Teile. Dabei wird die Aufgabenstellung so zergliedert, daß die Schnittstellen zwischen den Komponenten möglichst einfach zu gestalten sind. Nur wenn das Zusammenwirken der Teile sinnvoll geregelt ist, kann ein vernünftiges Ergebnis sowohl unter dem Gesichtspunkt der Implementierung als auch des Betriebs erreicht werden.

Die Probleme, die auch bei der Anforderungsdefinition angesprochen wurden, lassen sich auch auf den Entwurf übertragen. Um Mißverständnissen und Fehlinterpretationen in der Implementierungsphase vorzubeugen, sollte das Resultat des Entwurfs, die

Spezifikation, möglichst präzise formuliert werden. Darum sind formale Sprachmittel nötig, die eine automatisierte Behandlung der Spezifikation unterstützen.

Da Fehler in den frühen Phasen der Realisierung im allgemeinen erhebliche Kosten verursachen, wenn sie zu spät entdeckt werden, ist eine sehr sorgfältige Prüfung jedes Entwurfschrittes gerade in diesem Stadium besonders wichtig.

In /Ram80/ wird gezeigt, wie Fehler in der Anforderungsdefinition und in den frühen Phasen des Entwurfs gefunden werden können.

Neben den zuvor bereits erwähnten Werkzeugen werden beim Design eingesetzt Jackson-Diagramme, Entscheidungstabellen, SPECIAL und verwandte Methoden.

Bei der Jackson-Methode (/Jac75/) legt man Ein- und Ausgangsdaten zuerst fest. Durch die Kontrollstrukturen Sequenz, Selektion, Wiederholung und durch schrittweise Verfeinerung werden die Datenstrukturen definiert. Aus Überlagerung von Ein- und Ausgabestruktur versucht man dann, das Verarbeitungsdiagramm zu gewinnen.

Durch Entscheidungstabellen werden eine Menge von Bedingungen und Aktionen miteinander verknüpft. Das geschieht zweistufig. Zuerst gibt man an, welche Elementarbedingungen mit welchen Wahrheitswerten belegt sein müssen, damit eine Menge von Aktionen ausgeführt wird. Welche Funktionen dann aktiviert werden, zeigt die zweite Matrix an.

Ein Modul wird bei SPECIAL (/Rob77/) oder verwandten Sprachen in eine Menge von O- und V-Funktionen unterteilt. Mit den V-Funktionen wird dem Aufrufer ein Bild vom Zustand des Moduls vermittelt. Die Wirkung der O-Funktionen kann durch Veränderungen der V-Funktionen dargestellt werden. Da aber durch die V-Funktionen meist nur ein unvollständiges Bild

Über den Zustand vermittelt wird, führt man "versteckte" (hidden) V-Funktionen ein, die nicht sofort sichtbare Auswirkungen von O-Funktionen (delayed effects) darstellen. Auf diese Weise lassen sich alle O-Funktionen hinreichend präzise erläutern. Die "versteckten" V-Funktionen sind nicht zu implementieren.

Anders als bei SPECIAL und verwandten Spezifikationssprachen werden bei der Spezifikation mit abstrakter Repräsentation (/Ker82/) die Datenstrukturen nicht über den Umweg versteckter V-Funktionen eingeführt, sondern direkt als abstrakte Repräsentation des Objektes angegeben. Über bereits bekannte Datentypen definiert man damit ein neues Element.

Neben dem Zusammenspiel der Komponenten sollte im Entwurf auch untersucht werden, ob oder inwieweit die gestellten Anforderungen zu erfüllen und wie einzelne Komponenten und das Gesamtsystem zu testen sind.

Trotz aller Formalismen zum Zweck einer automatisierten Behandlung darf man nicht vergessen, daß eine Spezifikation in erster Linie der Kommunikation dient. Aus diesem Grund ist sie möglichst übersichtlich und präzise zu gestalten. Die Überlegungen bezüglich der Darstellungsweise sind ähnlich denen bei der Anforderungsdefinition.

3) Implementierung

In dieser Phase wird das Produkt so aufbereitet, daß es als Lösung des Problems akzeptiert werden kann. Am Ende der Implementierung steht der Integrationstest. Dabei stellt sich heraus, ob die Komponenten so zusammenwirken, wie es in der Spezifikation beschrieben ist. Erst in dieser Phase kann umfassend getestet werden, und hier entscheidet

sich, wie gut bei der Anforderungsdefinition und Spezifikation die Entwicklungsziele aufbereitet worden sind.

Schwierigkeiten bereitet es, Änderungen in der Spezifikation und den Anforderungen in die laufende Implementierungsarbeit einzubringen. Deshalb sollte man kritische Teile, in denen sich leicht etwas ändern kann, schon von Anfang an besonders kennzeichnen und dann so konzipieren und implementieren, daß mögliche Änderungen leicht zu realisieren sind.

Auf einzelne Sprachen einzugehen, würde den Rahmen dieses Überblicks sprengen. Ebenso wichtig wie die Programmiersprache sind in dieser Phase Werkzeuge, die den Entwicklungsprozeß unterstützen. Die Kontrolle über die verschiedenen Versionen eines Softwareproduktes und dessen Komponenten, sowie Test und Dokumentation sollte man durch geeignete Hilfsmittel automatisieren.

Eine gute Teststrategie ist eine der wichtigsten Voraussetzungen für einen effizienten Integrations-test. Prüfdaten dürfen nicht durch die Implementierer festgelegt werden. Bei einer neuen Version sind mindestens die Tests durchzuführen, die auch bei ihrem Vorgänger angewandt wurden.

4) Inbetriebnahme und Pflege

Nach der Implementierung muß das Softwareprodukt in seine spätere Umgebung eingebettet werden. Nun erkennt der Auftraggeber, ob das System leistet, was er sich davon versprochen hat.

Dabei wird es sich nur in den seltensten Fällen vermeiden lassen, daß sich das Produkt nicht mehr ändert.

Auch das unterstreicht die Notwendigkeit, nachträglich die Anforderungsdefinition, die Spezifikation

und die Implementierung modifizieren zu können. Auch nach der Inbetriebnahme sind die Produkte Änderungen unterworfen. Solange sie eingesetzt werden, entdeckt man Mängel (/Gla81/) oder wünscht sich funktionelle Erweiterungen und Anpassungen an neue Systeme. Modifikationen sind manchmal nur schwer in das bestehende Gefüge einzubringen. Das führt dazu, daß sich die ursprüngliche Konzeption immer weiter vom aktuellen Einsatzzweck entfernt. Dadurch wächst aber auch der Aufwand für die Pflege und das System wird immer ineffizienter. Früher oder später gelangt man zu jenem Punkt, wo es sich nicht mehr lohnt, an dem bestehenden System etwas zu ändern. Damit beginnt dann ein neuer Softwareentwicklungszyklus.

4.2 Methoden zur Verbesserung der Qualität während des Entwicklungszyklus

Unzulänglichkeiten der beteiligten Menschen und der eingesetzten Werkzeuge führen dazu, daß die Softwareprodukte nicht die Erwartungen der Auftraggeber erfüllen. Diese Mängel dann zu beseitigen, verursacht Kosten. Da die Behebung von Defekten aufwendiger wird, je länger sie unentdeckt bleiben, ist es besonders wichtig, Fehler so weit es geht einzudämmen und möglichst schnell zu finden.

Aus diesem Grund gehört zu jedem Schritt eine Kontrolle der gewonnenen Resultate. Alle Ergebnisse sollen nachvollziehbar sein.

Formale Beweise können zwar die Gültigkeit von Zwischenschritten belegen, jedoch sind nicht alle Vorgänge effizient auf dieser Basis nachzuvollziehen. Manchmal geht man dazu über, statt einer Verifikation nur eine umfangreiche Überprüfung (Validation) zu unternehmen. Dabei können Fehlerstatistiken und Untersu-

chungen über Fehlerquellen bei ähnlichen Projekten wertvolle Dienste leisten. Leider gibt es nur von wenigen Anwendungen verlässliche Unterlagen über die aufgetretenen Defekte. Ausnahmen wie /Gla81/, /Ram81/ und /Tra80/ können diese Regel nur bestätigen. Häufig werden die Schwierigkeiten nachträglich verharmlost oder nur recht nachlässig dokumentiert. Die Erfahrung ist heute immer noch die beste Quelle, um mögliche Defekte leicht herauszufinden.

Eine andere Methode zur Verbesserung der Qualität eines Softwareproduktes ist eine präzise Behandlung des Problems im frühen Stadium. Auf die möglichen Konflikte zwischen formaler Behandlung und Verständlichkeit weist bereits der vorige Abschnitt hin.

Die bisher angesprochenen Maßnahmen werden im wesentlichen vor der Inbetriebnahme wirksam. Doch auch während des Betriebs will man den verbleibenden Defekten begegnen können.

Die Maßnahmen zur "dynamischen Fehlerbehandlung" lassen sich wie folgt gliedern:

1) Redundanz

Zur Behandlung von Entwurfsfehlern können nicht identische Komponenten benutzt werden, wie dies bei den Hardwareredundanztechniken üblich ist. Es muß also mehrere Realisierungen geben, die je nach Anforderung parallel ablaufen oder nach Bedarf eingesetzt werden. Eine andere Möglichkeit ist, Redundanz in die Datenstrukturen einzubetten und auf diese Art und Weise fehlerbehaftete Einzelschritte zu maskieren.

2) Fehlerreaktionsroutinen

Beim Auftreten bestimmter abnormaler Ereignisse kann in eigens dafür vorgesehene Routinen verzweigt werden. Dort ist dann für diesen Fall eine spezielle Behandlung durchgeführt. Zusätzlich dazu sind

bestimmte Bereinigungsmaßnahmen durchzuführen.

3) Laufzeitkontrollen

Wenn an einem bestimmten Punkt im Programm eine Bedingung erfüllt sein soll, die sich nicht leicht statisch prüfen läßt, muß diese Kontrolle in den Code eingebracht werden. In manchen Fällen lagert man die Überprüfung in die Hardware ein (z.B. bei Speicherzugriffen).

4) Ereignisprotokoll (Log)

Alle wichtigen Ereignisse während des Ablaufs sollten möglichst chronologisch festgehalten werden. So läßt sich nach einem Ausfall leichter feststellen, was vorher gerade abgelaufen ist. Spezifische Programme können dieses "Log" auswerten und in manchen Fällen vielleicht einen gültigen konsistenten Systemzustand rekonstruieren.

Wie man Fehlerbehandlungstechniken sinnvoll in den Ablauf einbettet, wird in den nächsten Kapiteln weiter behandelt. Hier soll noch näher betrachtet werden, wie diese Methoden - allen voran die Redundanztechnik - zum Aufdecken von Entwurfs- und Implementierungsfehler herangezogen werden können.

4.3 Recovery-Blöcke

Fehler im Entwurf und in der Implementierung aufzudecken ist deshalb so schwer, weil man nicht weiß, wo und wie die Defekte eingepflanzt wurden. Hat man ihn erst einmal entdeckt, so ist die Beseitigung oft nur eine Kleinigkeit. Es gibt aber keinen Algorithmus, der in der Lage wäre, diese Fehler aufspüren. Deshalb bleibt nur der Weg übrig, durch Redundanz ein besseres Verhalten zu erzielen, nachdem alle Maßnah-

men zur statischen Fehleranalyse erschöpft sind. Wird eine Problemstellung mehrfach realisiert, um Defekte zu maskieren, so müssen alle Verwirklichungen möglichst verschieden (diversitär) sein, um Fehler mit hoher Wahrscheinlichkeit zu vermeiden. Wenn sich Bereiche überschneiden, kann ein Defekt, der in mehreren Realisierungen steckt, nicht immer aufgedeckt werden. Die Qualität dieses Ansatzes steht und fällt damit, wie stark die Überdeckung (coverage) der einzelnen Teile ist.

Ein weiterer Gesichtspunkt bei diesen Strategien ist, daß sie die Aufgabenlösungen nicht weiter unnötig erschweren dürfen. Ist diese Prämisse nicht erfüllt, so kann die zusätzliche Steigerung der Komplexität zu einem weiteren Verlust an Zuverlässigkeit führen. Eine Möglichkeit zur Behandlung von Fehlern, die keinen hohen zusätzlichen Aufwand in manchen Beziehungen erfordert, ist der Recovery-Block, wie er in /Ran75/ vorgestellt wurde:

```
ensure      "Abschlußtest"  
  
by         "Alternative 1"  
  
else by    "Alternative 2"  
  
...  
  
else by    "Alternative n"  
  
else error
```

Zuerst wird der Programmabschnitt "Alternative 1" ausgeführt, dann der "Abschlußtest" aufgerufen. Liefert er ein negatives Ergebnis, so werden sämtliche Zustandsvariablen in den Ausgangszustand zurückversetzt. In

diesem Fall versucht man "Alternative 2" und führt danach wieder der "Abschlußtest" durch, um festzustellen, ob nun ein befriedigendes Ergebnis erreicht wurde. Dieses Verfahren wird solange wiederholt, bis der Test ein positives Ergebnis liefert, oder die letzte Alternative erfolglos durchlaufen wurde. Konnte keine der Realisierungen ein befriedigendes Ergebnis liefern, so wird dem aufrufenden Programm signalisiert, daß ein Fehler vorliegt und der Auftrag nicht ausgeführt werden konnte.

Recovery-Blöcke verhalten sich wie gewöhnliche Blöcke in Programmiersprachen. In den einzelnen Alternativen können weitere Recovery-Blöcke benutzt werden.

Soll die Methode gut funktionieren, müssen zwei Bedingungen erfüllt sein:

- Die Alternativen müssen, wie vorher schon erwähnt, möglichst unabhängig voneinander zu implementieren sein.
- Der Abschlußtest muß in einem hohen Maße richtige Ergebnisse liefern, da die Zuverlässigkeit des Ganzen stark von dieser Routine abhängt. Kritisch ist vor allen Dingen, wenn die Überprüfungen bei einer fehlerhaften Alternative zu dem Resultat "richtig" kommt. Es ist aber schwierig, Abschlußtests so zu formulieren, daß alle möglichen Fehler aufgedeckt werden können.

Die erste Bedingung erfordert auch, daß die Spezifikation keine Implementierung besonders bevorzugt, da sonst diese Unabhängigkeit nur schwer zu realisieren ist.

Zum dynamischen Aufwand für den Recovery-Block-Mechanismus gehört neben dem Abschlußtest noch das Retten der Zustandsvariablen, um vor Eintritt in eine weitere Alternative den ursprünglichen Zustand wieder-

herzustellen. Man kann zeigen, daß dieser Aufwand proportional zur Zahl der modifizierten Variablen ist. Das bedeutet, daß bei Änderungen einzelner Elemente einer großen Datenbasis keine Kopie des gesamten Objektes erstellt werden muß. Auch das Rekonstruieren des alten Zustands ist ebenfalls mit linearem Aufwand bezüglich der im Recovery-Block modifizierten Variablen zu bewerkstelligen. Probleme im Zusammenhang mit dem Sichern der Zustandsvariablen erörtert das Kapitel über das Transaktionsmodell ausführlich.

Ein großer Vorteil der Recovery-Block-Methode ist, daß nicht auf einzelne Fehlerarten und -orte eingegangen werden muß. So kann man eine Vielzahl von Fehlern auf ein und dieselbe Weise erkennen und beheben. Die Fehlerarten, die erfaßt werden, lassen sich durch Analyse des Abschlußtests bestimmen. Dies zeigt jedoch deutlich auf, wie schwierig eine optimale Gestaltung der Prüfroutine ist. Einerseits soll sie einfach sein, damit sie möglichst fehlerfrei und ohne Belastung des Betriebs realisiert werden kann, andererseits soll der Test so umfassend sein, daß die schwerwiegendsten, mangelhaften Ausführungen von Alternativen erkannt werden.

Erschwert wird diese Aufgabe zusätzlich noch durch die Bestimmung der Fehleranfälligkeit der einzelnen Alternativen. Damit sind wir an einem Punkt angelangt, an dem die Grenzen dieser Methode deutlich identifizierbar werden. Positiv zu vermerken ist noch, daß bei dieser Methode die Fehlererkennung zentralisiert werden kann.

Eine weitere Voraussetzung für den Einsatz des Recovery-Block-Mechanismus ist, daß die Zustandsvariablen reversibel sein müssen, d.h. auf ihren vorhergehenden Wert gesetzt werden können müssen. Dies ist zwar bei den Speichermedien erfüllt, nicht aber bei sogenannten externen Modulen (z.B. beim Modul "Drucker" die

Zustandsvariable "gedruckte Zeichen"). Doch auch hier bieten sich meist sinnvolle Lösungsmöglichkeiten an. So kann eine Druckerseite als fehlerhaft markiert, dies dem Operateur mitgeteilt und die Seite wiederholt werden.

4.4 N-Version-Programmierung

Die für manche Anwendungsfälle noch zu geringe Diversität des Recovery-Block-Konzeptes führte zur Entwicklung der N-Version-Programmierung (/Che78/, /Tra80/). Dabei lassen sich verschiedene Programmiersprachen zur Lösung des gleichen Problems einsetzen, vor allem aber können die verschiedenen Realisierungen des Produktes auf unterschiedlicher Hardware ablaufen. Da Ereignisse erst dann überprüft werden müssen, wenn sie der Umwelt sichtbar gemacht werden, entstehen auf den ersten Blick keine Probleme durch Rücksetzen von Anwendungen. Kein Aufwand ist nötig, um die Abbilder der Datenstrukturen vor der Modifikation zu retten. Das jedenfalls führen die Befürworter dieser Methode ins Feld.

In ihrer Vorgehensweise ist die N-Version-Programmierung der maskierenden Hardwareredundanz sehr ähnlich, während das Recovery-Block-Konzept mit der dynamischen Redundanz Gemeinsamkeiten aufweist. Damit ist schon erkennbar, welche Nachteile man bei diesen Arten von Fehlerbehandlung in Kauf nimmt, z.B. das Wartungsproblem. Was passiert, wenn eine der Versionen ausfällt, d.h. ein verkehrtes Ergebnis liefert? Wie bei der statischen Redundanz verschlechtert sich die Zuverlässigkeit mit wachsender Betriebsdauer, wenn keine Pflege des Systems durchgeführt wird.

Wie aber läßt sich die inkonsistente Datenstruktur eines defekten Moduls reparieren und in die laufende Bearbeitung wieder einbringen? Wenn in jeder Version

andere Datenstrukturen benutzt werden, wie das ja wegen der Diversität bei der N-Version-Programmierung erwünscht ist, dann ist eine Aktualisierung während des laufenden Betriebes nur mit großem Aufwand möglich. Die Missionszeit eines so entworfenen Systems muß also der Zuverlässigkeit der Versionen angepaßt sein.

Da es sich bei den eingesetzten Rechenanlagen um sehr komplexe, verschiedenartig aufgebaute Gebilde handelt, ist das Synchronisieren der einzelnen Schritte und der Vergleich der gewonnenen Ergebnisse mitunter sehr schwierig. Noch leichter als bei der statischen Hardwareredundanz kann der Fall eintreten, daß sich die einzelnen Komponenten "aussynchronisieren", d.h. Bearbeitungsreihenfolgen verändert werden. Es ist nur schwer auszuschließen, daß die einzelnen Versionen den Eingabedatenstrom in anderen Relationen sehen. Rundungsfehler oder Ausnutzung von bestehenden Freiheitsgraden bei der Realisierung erschweren es der Vergleichseinheit, die gravierenden Mängel von den vernachlässigbaren Abweichungen zu trennen.

Aus diesen Gründen setzt man diese Methode auch in einer leicht abgewandelten Form ein. Dabei dient sie ausschließlich dazu, Entwicklungsfehler vor oder während der Inbetriebnahme aufzudecken(/Ram79/). Es werden für dieselben Anforderungen mehrere Spezifikationen und Implementierungen entwickelt und mit den gleichen Datenmengen getestet. Gibt es Unterschiede in den Ergebnissen, so wird geprüft, ob die Differenzen vernachlässigbar sind oder welche Komponenten noch Fehler enthalten. Dadurch kann der Testaufwand so stark reduziert werden, daß sich die Mehrkosten für die doppelte Implementierung lohnen. Da auch die Spezifikation vervielfacht wird, kann schon am Ende dieser Phase manuell geprüft werden, inwieweit die verschiedenen Entwürfe übereinstimmen. Damit kann man schon früh-

zeitig Rückschlüsse auf die Anforderungen ziehen. Dieser "duale" Ansatz hat sich beim frühzeitigen Aufdecken von Defekten als sehr nützlich erwiesen, wenn sehr hohe Zuverlässigkeit gefordert ist und der Aufwand für die Beseitigung von Entwurfsfehlern klein gehalten werden muß.

4.5 Bewertung der einzelnen Verfahren

Wenn man die Konsequenzen aus den Erfahrungen mit der Entwicklung von Softwareprodukten zieht, so stellt sich heraus, daß die Methoden, die im Anfangsstadium Defekte aufspüren, am erfolgversprechendsten erscheinen.

Präzise und verständliche Formulierung der Anforderung und Spezifikation, kombiniert mit einer Schwachstellenanalyse (z.B. Revisionen, Walk-Through, Checklisten, Teststrategien) bilden die Grundlage für den Entwurf zuverlässiger Software (/Keu82/).

Mit Software-Redundanz kann man Entwurfsfehlern auch im Betrieb begegnen. Es gibt aber bei deren Anwendung noch eine Vielzahl ungelöster Probleme.

5. Recovery - Arten

Wenn irgendwo ein Defekt auftritt und erkannt wird, muß eine Reaktion erfolgen. Dabei unterscheiden sich die Maßnahmen zur Fehlerbehebung (Recovery) von Situation zu Situation.

Mögliche Reaktionen sind:

- Abbruch einzelner Programme oder Anwendungen
- Wiederholung von Aktivitäten
- Korrigieren des Zustands
- Umschalten auf redundante Komponenten
- Wiederanlauf mit einer Kopie des Systemzustand vor dem Absturz (Warmstart)
- Löschen aller während den vorangegangenen Aktivitäten gewonnenen Information und Neubeginn mit dem initialen Zustand (Kaltstart)

Dabei variiert das Maß, wie weit auf die aktuellen Zustände der Module zurückgegriffen wird. Bei Kaltstart spielt die momentane Situation keine Rolle, während bei Verbesserungsversuchen wie Wiederholen und Korrigieren weiter mit der aktuellen Datenbasis gearbeitet wird.

Einen Einfluß auf die Recoverymaßnahmen übt auch die Art aus, wie die Nachricht von einem Defekt auf die Komponenten trifft. Ist sie einer bestimmten Aktion fest zuzuordnen, der größte Teil der Datenstrukturen trotzdem noch intakt und sind die Auswirkungen des Defektes gering, so ist eine Bearbeitung noch relativ

einfach. Kann man aber nicht von konsistenten Datenbeständen ausgehen und ist die Fehlermeldung unkorreliert zu der laufenden Bearbeitung, so findet sich nur schwer eine sinnvolle Recoverymaßnahme.

Nachrichten über Defekte können auf zwei Wege zu einem Modul gelangen, wenn man ein hierarchisch strukturiertes System zugrunde legt:

1) Als Rückmeldung unterlagerter Komponenten

Dies ist der gebräuchlichere Weg, wenn unterlagerte Bausteine Mängel aufweisen (z.B. Hardwarefehler) oder Programme der höheren Schicht Dienstleistungen anfordern, die nicht realisiert sind, die im momentanen Zustand nicht verwendet werden dürfen oder deren Eingangsparameter falsch sind.

2) Als Anforderungen von höheren Ebenen

Stellt sich heraus, daß eine Teilaktivität nicht erfolgreich abgeschlossen werden kann, so muß oft ein ganzer Vorgang abgebrochen und vorangehende Operationen rückgängig gemacht werden. Wenn also ein unterlagertes Modul einen Fehler meldet, so kann das auch bisher erfolgreich abgeschlossene Aktivitäten in anderen Komponenten betreffen. Man wird also diese Module benachrichtigen, um mit ihrer Hilfe unerwünschte Effekte zu beseitigen.

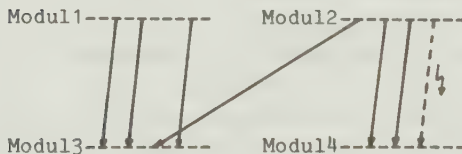


Abbildung 5.1:

Recoveryausbreitung auf unterlagerte Module,
die von mehreren Komponenten benutzt werden

Aus den verschiedenartigen Anforderungen läßt sich ableiten, daß es wohl keine einfache Methode gibt, die alle Fälle gleich gut abdeckt. Fehlern, die von unteren Schichten gemeldet werden, begegnet man in den meisten Fällen durch individuelle Reaktion auf das Ereignis, da hier oft der größte Teil des Moduls noch konsistent geblieben ist. Anders verhält es sich, wenn man die Fehlerbehandlung auf intakte Komponenten ausbreiten muß, die von mehreren Modulen benutzt werden (Abbildung 5.1). Dabei kann es vorkommen, daß auch Aktivitäten, die eigentlich nichts mit dem Defekt zu tun haben, betroffen sind. Eine individuelle Bearbeitung aller möglichen Aufrufsequenzen im Fehlerfall führt sehr schnell zu kombinatorischen Problemen.

In den nächsten Abschnitten werden deshalb verschiedene Lösungsansätze für die Recovery-Probleme dargestellt und miteinander verglichen.

5.1 Unerwünschte Ereignisse

Im vorangegangenen Kapitel wurden bereits zwei Methoden angesprochen, die eine dynamische Behandlung von Defekten erlauben - das Recovery-Block-Konzept und die N-Version-Programmierung. Schwierigkeiten bei deren Anwendung sind

- Finden von alternativen Implementierungen, die möglichst wenige Gemeinsamkeiten haben (Diversität),
- Gestaltung des Akzeptanztests,
- Bearbeitung von Datenstrukturen mehrerer paralleler Prozesse (dieser Aspekt wird noch eingehend behandelt),

- Rücksetzbarkeit von Zustandsvariablen

usw.

Bei dem Recovery-Block-Konzept nimmt man zwangsläufig in Kauf, daß bei Fehlern unter Umständen sehr viele Aktivitäten rückgängig gemacht werden, die eigentlich korrekt durchgeführt worden sind. Zusätzlich kommt durch das Retten der Zustandsvariablen und durch den Akzeptanztest noch ein Aufwand hinzu, der auch im fehlerfreien Zustand das System stark belastet.

Kann man davon ausgehen, daß die Fehlerquellen bekannt sind und die Defekte relativ häufig auftreten, so ist diese Art der Fehlerbehandlung nicht leistungsfähig genug. Bei diesen "kalkulierbaren", aber nicht statisch zu beseitigenden Mängeln (sie sollen zur besseren Unterscheidung nach Wür77/ als "unerwünschte Ereignisse" bezeichnet werden), ist der Aufwand oft beträchtlich zu reduzieren. So kann man bei einer Division durch Null als Ergebnis den größtmöglichen darstellbaren Wert annehmen und damit das Programm fortsetzen. Bei Paritätsfehlern in Speichern genügt meist ein weiterer Lese- oder Schreibvorgang.

Sind also solche Möglichkeiten bekannt, so versucht man, sie schon beim Entwurf mit zu berücksichtigen. Zu beachten ist in dieser Phase vor allem, daß bei der Zerlegung des Systems in Module die Recovery mit-einzubeziehen ist, da durch die Abstraktion an den Schnittstellen der Objekte für die Fehlerbearbeitung wichtige Informationen verlorengehen können. Ein Beispiel für unzureichende Fehlerinformation ist die bei vielen technischen Geräten anzutreffende "Stopplampe". Die Gründe für den Ausfall bleiben dabei dem Anwender verborgen, gleichzeitig aber wird durch diese Vereinfachung eine schnelle Behebung der Fehlerursache oft erheblich erschwert.

Um eine Behandlung von unerwünschten Ereignissen sinn-

voll durchzuführen, ist eine Unterstützung durch geeignete Mechanismen und deren Abbildung auf Sprachmittel der Programmiersprachen nötig.

Erste Ansätze dieser Art wurden durch die "On-Conditions" in PL/1 (/Mel74/) eingearbeitet. Der wesentliche Effekt bei diesen On-Conditions ist, ein vom Benutzer zu definierendes Unterprogramm im Fehlerfall aufzurufen. Das geschieht mit einem interrupt-ähnlichen Mechanismus (Trap).

Nach der Behandlung fährt man wieder an der Stelle fort, an der das Ereignis auftrat (oder unmittelbar dahinter). Auf diese Weise läßt sich ein Fehler leicht in der Umgebung behandeln, wo er auftrat und die Operation kann vielleicht doch noch erfolgreich beendet werden. Die Rückkehr zu der fehlerverursachenden Stelle ist aber nur in den seltensten Fällen sinnvoll durchzuführen, da die "Normalroutinen" dafür nicht konzipiert sind. Es ist in der Sprachdefinition unter anderem auch vorgesehen, als Standardreaktion auf Fehlermeldungen einfach fortzufahren.

Eine in vielen Punkten genau entgegengesetzte Strategie verfolgte man in Ada (/Ich79/).

procedure Q is

begin

... P; ...

exception

when EX => Reaktion auf die in P definierte
Fehlerbedingung;

end

Abbildung 5.2:

Ada-Exception-Handling I

procedure P is

begin

EX: exception ;

... raise EX ; ...

exception

'vordefinierte Fehlerbedingungen

when NUMERIC-ERROR => Reaktion auf Überlauf,..

when CONSTRAINT-ERROR => R. bei Zugriffen außerhalb der Feldgrenzen

when SELECT-ERROR => R. wenn es keine offene "select"-Klausel gibt

when STORAGE-ERROR => R. wenn geforderter Speicherplatz nicht verfügbar

when TASKING-ERROR => R. bei fehlerhafter Task-synchronisation

end

Abbildung 5.3:

Ada-Exception-Handling II

Wenn ein Fehler entdeckt wird, verläßt man nach dem Durchlaufen des "Exception-Teiles" die Verarbeitungsroutine. Der gerade bearbeitete Block kann dann mit oder ohne Fehleranzeige an die aktivierende Schicht beendet werden. Wird ein unerwünschtes Ereignis im aufrufenden Programm nicht beachtet, so wird auch dieser Abschnitt verlassen und die Fehlerbedingung im umschließenden Block gesetzt. Auch Fehler im Zusammenspiel mehrerer Prozesse können von Ada behandelt werden (Tasking-Errors).

Eine weit verbreitete Art der Fehlerübergabe bei Un-

terprogrammen ist das Hinzufügen von Statusparametern. Damit kann der Aufrufende nach Abschluß der Operation feststellen, ob das Unterprogramm auch richtig durchlaufen wurde. Es ist zwar eine ähnliche Wirkung zu erzielen, man muß aber mit höherem Aufwand rechnen als bei den oben beschriebenen Mechanismen. Die Gefahr ist groß, daß der Benutzer diese Meldungen einfach ignoriert.

Um Fehlerbehandlung gut strukturieren zu können und keinen unnötigen Mehraufwand in der Programmierung leisten zu müssen, sollte man für die Behandlung von unerwünschten Ereignissen folgende Konstrukte vorsehen:

- Der Aufruf einer Operation kann mit Routinen versorgt werden, die auf Fehler reagieren sollen
- Vom Benutzer und vom Implementierer eines Moduls soll der Ablauf der Fehlerreaktion zu steuern sein. Als Sprachkonstrukte (/Goo75/) dafür bieten sich an:

ESCAPE zeigt dem Aufrufer an, daß eine Fortsetzung der Operation nicht möglich ist

SIGNAL stellt dem Anwender frei, ob er nach der Fehlerbehandlung zur unterbrochenen Ope-

ration zurückkehren will (durch RESUME) oder hinter der aufgerufenen Funktion fortfahren will (durch EXIT). Mit CLEANUP kann innerhalb eines Moduls definiert werden, was zu tun ist, wenn nach einem SIGNAL die Fehlerbehandlungsroutine durch ein EXIT verlassen wird.

NOTIFY dient nur zur Benachrichtigung des Aufrufers. Die Operation, die dieses uner-

wünschte Ereignis erkannt hat, wird aber danach normal weitergeführt.

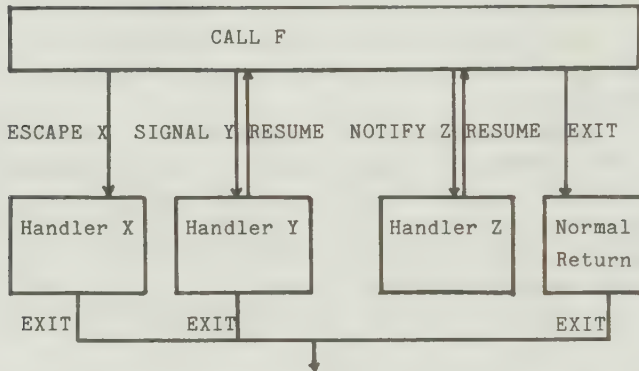


Abbildung 5.4:
Exception-Handling nach /Goo75/

- Standardfehlerbehandlung festzulegen, nicht nur vom System aus, sondern auch von den Implementierern her für ihren Bereich.
- Hierarchien von Operationen, unerwünschten Ereignissen, Handlern und Benachrichtigungsschemata festzulegen. Meist geht man hier von der Aufrufhierarchie aus. Fehlerbehandlung durch andere Prozesse kann aber notwendig werden, vor allem, wenn parallele Aktivitäten den Defekt heraufbeschworen haben (diese Feststellung wird in den folgenden Kapiteln noch näher erläutert).

Zwar ist es möglich, diese Ideen auch ohne die Unterstützung durch Programmiersprachen zu verwirklichen, jedoch kann dann der Programmierer nicht vom Übersetzer und Binder unterstützt werden.

Folgende Fehler sind feststellbar:

- ob der Programmierer vergessen hat, einem Aufruf die spezifischen Fehlerbehandlungsroutinen zuzuordnen,
- ob ein unerwünschtes Ereignis erwartet wird, das die aufgerufenen Module gar nicht erzeugen,
- ob nach einer ESCAPE-Exception mit RESUME oder bei einer NOTIFY-Exception mit EXIT abgeschlossen wird,
- u.ä.

Bei der Wiederaufnahme des normalen Betriebes ist es oft notwendig, eine nicht korrekt abgeschlossenen Operationen ganz zu wiederholen. Deshalb bietet es sich an, neben RESUME und EXIT noch RETRY zur Wiederholung der unterbrochenen Funktion vorzusehen.

Diese Methode der Fehlerbehandlung erweist sich zwar für eine eng begrenzte Zahl verschiedener Fehler als sinnvoll, belastet aber doch den Implementierer sehr stark, sobald die Zahl der möglichen Fehler überhand nimmt. Seiteneffekte von Fehlern in Aktivitäten, die parallel ausgeführt werden, lassen sich meist nicht so ohne weiteres beheben, wenn man sich bei der Fehlerbehandlung auf einen Prozeß beschränken muß.

Dies zeigt, daß auch der andere Weg, der die Fehlerbehandlung mehr zu automatisieren versucht, weiter verfolgt werden muß.

5.2 Rückwärts-Recovery

Während man bei der im vorigen Abschnitt beschriebenen Methode versucht, aus der aktuellen Situation heraus wieder in einen fehlerfreien Zustand zu gelangen, ist die Strategie bei der "Rückwärts-Recovery", sich auf

In der konkreten Realisierung lassen sich beide Vorgehensweisen nicht so scharf voneinander abgrenzen. Auch bei der Vorwärts-Recovery sind Änderungen rückgängig zu machen und Aktionen zu wiederholen (wie das "CLEANUP" und das "RETRY" zeigen) und bei der Rückwärts-Recovery werden sich manche Fehlerbehandlungsoperationen nicht von den normalen Aufrufen unterscheiden. Für die hier beschriebene Methode ist charakteristisch, daß man nicht auf die konkrete Fehlerursache eingeht, wie es bei dem Exception-Handling üblich ist, sondern versucht, möglichst viele Fehlerquellen auf eine Art und Weise zu behandeln. Es werden bei der Rückwärts-Recovery bestenfalls die Auswirkungen eines Defektes beseitigt, nicht aber dessen Ursache.

Eine Grundlage für Rückwärts-Recovery-Aktivitäten in fast allen realen Systemen ist die Idee, daß man den Zustandsvektor um Komponenten erweitert, die einen vorangegangenen, konsistenten Zustand repräsentieren können. Wenn man nun Fehler bald nach ihrem Auftreten entdeckt, und die dadurch verursachten Defekte keine Auswirkungen auf die Zusatzinformationen haben, so ist eine relativ optimale Recovery zu erzielen. Diese redundanten Informationen bilden das Rückgrat für alle Aktionen bei dieser Art von Fehlerbehandlung. Dabei ist abzuwägen, wieviel und welche Informationen dabei zusätzlich gesammelt werden sollen. Bearbeitungszeit und Speicherplatz für diese Daten können bei ungeschickter Wahl das System überlasten. Der Aufwand wächst zwar linear mit der Zahl der veränderten Variablen, kann aber erheblich reduziert werden, wenn man Lebensdauer und Zweck der einzelnen Größen mit berücksichtigt. So sind lokale Variablen vernachlässigbar, wenn sie nach der Recovery sowieso wieder auf den vorgegebenen Anfangswert gesetzt werden. Die anfallende Recovery-Information sollte auf einem "stabilen" Speichermedium abgelegt werden, damit diese

Daten die Defekte auch "überleben" (/Reu81/). Dazu kann man die in 4.2 erwähnte Ereignisprotokollierung verwenden. Diese Forderung aber bedingt, daß die Recovery-Daten zu bündeln sind, um einen schnellen Transfer zu diesen weniger leicht zugänglichen Speichern (Log) zu ermöglichen. Dadurch wird oft eine große Menge nicht veränderter Daten gesichert. Allgemein gilt: je weniger Platz man für die Abspeicherung der redundanten Daten aufwenden will, desto komplexer werden die Recovery-Algorithmen.

Die Forderung, daß Fehler möglichst bald entdeckt werden müssen, läßt sich auch aus dieser Grundidee ableiten. Je länger Defekte unentdeckt bleiben desto komplexer wird die Recovery und desto weiter entfernt man sich dem gewünschten Zielzustand.

Zum Schluß dieses Abschnitts soll noch einmal über die Rückwärts-Recovery-Aktion gesprochen werden. Nicht immer ist es mit vertretbarem Aufwand möglich, vor Beginn einer Aktion einen Abzug der Datenbasis zu machen und diese Kopie zu benutzen, falls ein Defekt auftritt. Meist werden nur die Werte von Variablen vor ihrer Veränderung ("Before Images") in das Log geschrieben. Eine andere Möglichkeit wäre, bei Fehlern die verursachende Aktion zu invertieren. Diese Maßnahme erfordert geringen Log-Aufwand und ist auch auf "irreversible Variable" (Ein-/Ausgaben) anwendbar. Es ist aber nicht hinreichend, einfach eine inverse Funktion als Recovery anzuwenden, wenn nicht festgestellt werden kann, ob eine Operation ganz, gar nicht oder teilweise ausgeführt wurde.

Bei Ausfällen ist immer damit zu rechnen, daß die Aktivitäten, die man rückgängig machen will, nicht vollständig abgeschlossen werden.

Zur besseren Erläuterung von Recoveryaktionen sollen folgende Bezeichnungen eingeführt werden:

- * Sei a eine Aktion, die durch Konkatenation von unterbrechungsfreien Elementaroperationen gebildet wird:

$$a = a_1 o \dots o a_n$$

- * Mit $a' = a_1' o \dots o a_n'$ wird die Recovery-Aktion zu a bezeichnet.

- * a, a' - bezeichnen Anfangsstücke von a, a' (einschließlich der leeren Sequenz und a, a')

$$a \in \{\Lambda, a_1, a_1 o a_2, \dots, a\}$$

$$a' \in \{\Lambda, a_1', a_1' o a_2', \dots, a'\}$$

- * id ist die Funktion, die eine Menge auf sich selbst abbildet

Definition 5.1:

Die (Rückwärts-)Recovery-Aktion a' zu a hat folgende Eigenschaften:

a) $id = a - o a' - o \dots o a' - o a'$

- b) der Log-Eintrag für a erfolgt unterbrechungsfrei vor der Durchführung von Veränderungen durch a (Write-Ahead-Log-Prinzip)

Bemerkungen zu 5.1

- Da Recovery-Aktionen sehr komplex werden können, ist die Wahrscheinlichkeit groß, daß während ihres Ablaufs Ausfälle auftreten. Das wirkt sich besonders aus in den Fällen, bei denen nach einem Fehler die Wahrscheinlichkeit eines weiteren Defektes hoch ist.
- Die hier beschriebenen Recovery-Aktionen gehen davon aus, daß Fehler keine Nebenwirkungen haben.

- Nur wenn vor Ausführung einer Aktion ins Log geschrieben wird, ist eine "Rückwärts-Recovery" möglich. Es gibt einen fundamentalen Zusammenhang zwischen der Komplexität der Rückwärts-Recovery-Aktion und dem Umfang der Log-Einträge. Wenn eine Berechnung in viele einfache Operationen zerlegt wird, so muß man mit vielen Log-Daten rechnen. Sind viele Aktivitäten zusammengefaßt, so steigt die Komplexität der "Rückwärts-Aktionen" stark an, während die Log-Einträge weniger werden.

Anhand des Beispiels "Warteschlange" läßt sich zeigen, daß die Anforderungen von Definition 5.1 nicht so ohne weiteres auf beliebigen Abstraktionsniveau sinnvoll zu realisieren sind. Man kann als a die Operation "Entfernen eines Elementes" und als a' "Einfügen des alten Eintrags an erster Stelle" nehmen. Nach einer Recovery-Aktion ist dann das Element doppelt vorhanden, wenn der Log-Eintrag durchgeführt wurde, ohne daß das Element tatsächlich entfernt worden ist.

Es gibt verschiedene Methoden, Recovery-Aktionen zu realisieren. Relativ einfache Verfahren lassen sich anwenden, wenn zu Beginn einer Aktion leicht festgestellt werden kann, welche Größen sich in der Operation verändern. Als Recovery-Aktion braucht man in diesem Fall nur die Variable auf das "Before-Image" zu setzen, den Wert der Variablen vor der Modifikation. Darum versucht man auch in vielen Anwendungen der Rückwärts-Recovery, sich auf diesen einfachen Fall zurückzuziehen. Die Recovery wird komplizierter, wenn die zu modifizierenden Größen dynamisch zu ermitteln sind, oder außerhalb dieser Operation schwer zugänglich sind. Dies ist zum Beispiel der Fall, wenn innerhalb einer Operation eine Nachricht an einen anderen Rechner gesendet wird.

5.3 Recovery und parallele Aktivitäten

Recovery-Block-Konzept und ähnliche Verfahren zur Fehlerbehandlung lassen sich leider nicht so ohne weiteres auf parallele Prozesse übertragen. Nur wenn diese isoliert voneinander arbeiten, kann man es voll übernehmen.

Wenn aber Prozesse miteinander kooperieren und Informationen untereinander austauschen, muß auch die Fehlerbehandlung aufeinander abgestimmt sein.

Welche Folgen eine unkoordinierte Fehlerbehandlung hervorruft, wenn Prozesse miteinander kooperieren, kann am besten anhand des Domino-Effektes veranschaulicht werden. Der Domino-Effekt kennzeichnet Fehlerfälle, die einen Neustart des gesamten Systems bewirken, obwohl Recovery-Mechanismen eingesetzt werden. Abbildung 5.5 zeigt eine solche Situation. Waagrechte Linien deuten die Prozesse an. Senkrechte Linien beschreiben die Kommunikation. Der Übersichtlichkeit halber wurde als Synchronisationsmechanismus die Rendezvous-Technik herangezogen (/Ich79/). Die Klammern repräsentieren einen Recovery-Punkt - diese Zustände der Prozesse sind jederzeit wiederherstellbar (z.B. der Anfang eines Recovery-Blockes).

Wenn nun in Abbildung 5.5 an der Stelle t Prozeß P2 einen Fehler entdeckt, so muß das gesamte Prozeßsystem wieder auf den Startzustand zurückversetzt werden. Bei P2 muß c5, der letzte Recovery-Punkt, wiederhergestellt werden. Davon sind auch die Rendezvous r5 und r10 betroffen, weil für den Kommunikationspartner P2 die empfangenen und abgegebenen Informationen nicht mehr gültig sind. Also müssen alle Aktionen von P1 nach c2 und von P3 nach c8 gelöscht werden. Damit muß r4 zurückgenommen werden. Das wiederum bewirkt ein Rücksetzen von Prozeß P2 (und zwar auf c4). usw.

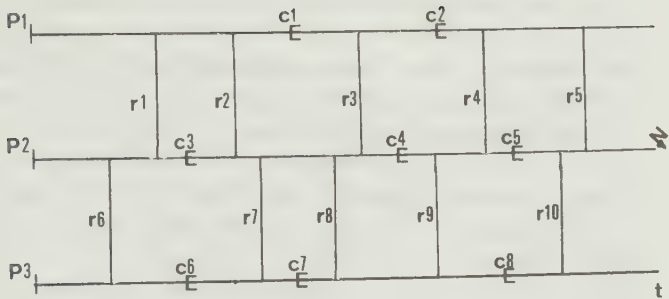


Abbildung 5.5:
Domino-Effekt

Die Folgen dieses Domino-Effektes:

- In Realzeitanwendungen muß die Fehlerbehandlung in definierter Zeit erfolgen und darf nur einen überschaubaren Teil der vorangegangenen Aktivitäten betreffen. Das ist aber so nicht zu garantieren.
- Zu keinem Zeitpunkt kann eine Aktion als abgeschlossen gelten. Selbst beendete Prozesse können aufgrund von Fehlern erneut gestartet werden müssen.

Der Domino-Effekt kann bei jeder Art von Rückwärts-Recovery und bei fast jeder Art von Synchronisation auftreten. Bei Mechanismen, die fehlerhafte Aktionen kompensieren (siehe Abschnitt 5.1), tritt dieser Effekt per Definition nicht auf, jedoch wird man dort mit einer hohen Komplexität der Algorithmen zur Fehlerbehandlung und mit einer entsprechend langen Laufzeit rechnen müssen. Dies kann man als eine Art "domino-Effekt" in Vorwärtsrichtung ansehen.

Der Domino-Effekt wird nur dann wirksam, wenn zwei Bedingungen erfüllt sind:

- Die Recovery-Punkte der einzelnen Prozesse sind nicht auf den Nachrichtenaustausch abgestimmt
- Es existiert eine geschlossene Kette von Prozessen P_i , für die gilt: wenn P_i rückgesetzt wird, so auch dessen unmittelbarer Nachfolger.

Will man den Domino-Effekt vermeiden, so muß man dafür sorgen, daß eine dieser beiden Bedingungen nicht erfüllt ist.

Es gibt eine Reihe von Möglichkeiten, die erste Bedingung zu umgehen. Auf die zweite Bedingung kann man wohl nur in Spezialfällen in einem System verzichten. Die primitivste Methode, um die Recovery-Punkte zu koordinieren, ist, wenn alle Prozesse gleichzeitig ihre Recovery-Punkte setzen (System Checkpoint). Damit ergeben sich aber weitere Schwierigkeiten. So ist dabei problematisch, diesen Ruhepunkt bei einem verteilten System zu erzwingen. Der Aufwand wächst, weil man eventuell auch lokale Zustandsvariable retten muß. Dadurch muß der gesamte Zustand eines Prozesses einschließlich der Registerbelegung und der temporären Variablen gerettet werden. Die Parallelarbeit kann dadurch stark eingeschränkt werden und so die Effizienz des Gesamtsystems stark sinken. Normalerweise ist bei jedem Fehler eine Rückkehr zum letzten System Checkpoint notwendig. Dabei können auch viele richtig abgeschlossene Aktivitäten betroffen sein. Damit keine Probleme beim Rücksetzen entstehen, muß gewährleistet sein, daß das System zur Zeit des Checkpoints in einem fehlerfreien Zustand ist. Nicht durchführbar ist diese Strategie, wenn Aktivitäten in relativ kurzen Zeiträumen abgewickelt werden müssen, da ein Checkpoint im allgemeinen alle Aktivitäten für längere Zeit lahmlegt.

Eine etwas verfeinerte Art, um dem Domino-Effekt zu

begegnen, nennt man "Konversation" (/Ran75/). Bei dieser Methode müssen nur die Recovery-Punkte der an der Konversation beteiligten Prozesse synchronisiert werden. Unbeteiligte Prozesse können anders als beim systemweiten Ruhepunkt ohne Verzögerungen weiterarbeiten. Voraussetzung dafür ist, daß der Kreis der Teilnehmer an einer Konversation bekannt ist und deren Dauer sich beschränken läßt. Sonst ist die Gefahr hoch, daß während ihres Ablaufes Fehler produziert werden und dadurch viel Information verlorenggeht. Prozesse können keine Konversation beginnen, wenn die letzte nicht beendet ist.

Während einer Konversation darf es keine Interaktionen mit unbeteiligten Prozessen geben.

Einen weiteren Ansatz, die Ausbreitung von Fehlern in den Griff zu bekommen, machte man mit den Recoverylinien (/Ran79/).

Eine Recovery-Linie RL ist eine Menge von Recovery-Punkten, für die gilt:

- a) ein Recovery-Punkt r unterteilt eine Prozeß P_i in $\text{Pre}(P_i, r)$ - die Aktivitäten von P_i vor r und $\text{Post}(P_i, r)$ - die Operationen von P_i nach r .
- b) Die Recovery-Punkte aus RL stammen alle aus verschiedenen P_i .
- c) Es gibt keine Interaktion zwischen einem Prozeß, der keinen Recovery-Punkt in RL hat und einer Operation aus $\text{Post}(P_i, r)$, wenn r Element von RL ist.
- d) Es gibt keine Interaktion zwischen Elementen aus $\text{Post}(P_i, r_i)$ und $\text{Pre}(P_j, r_j)$ oder $\text{Post}(P_j, r_j)$ und $\text{Pre}(P_i, r_i)$, wenn r_i, r_j aus RL.

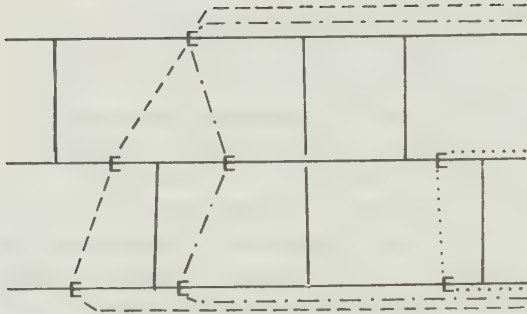


Abbildung 5.6:
Recovery-Linien

Auch System Checkpoints können als Recovery-Linien betrachtet werden. Bei Konversationen haben alle daran beteiligte Prozesse eine gemeinsame Recovery-Linie, wenn sie alle vor Beginn der Konversation einen Recovery-Punkt erzeugen.

Ein Recovery-Punkt kann gleichzeitig Mitglied mehrerer Recovery-Linien sein. Durch sie werden eine Menge von Abläufen isoliert, so daß bei Fehlern maximal die beteiligten Prozesse betroffen sein können. Die Operationen von $\text{Post}(P_i, r)$ müssen in diesem Fall eventuell zurückgesetzt werden.

Anders als bei der Konversation brauchen hier die an einer Recovery-Linie beteiligten Prozesse nicht bekannt zu sein, es muß nur eine Stelle im System geben, die überprüft, ob bei einer Kommunikation bzw. beim Löschen eines Recovery-Punktes die letzte Recovery-Linie verlorenggeht. Jedoch bedingt diese Vorgehensweise eine zentrale Kontrolle der Recovery-Punkte. Das macht dieses Verfahren für verteilte Systeme nur bedingt einsetzbar.

Das Aufgeben einer Recovery-Linie RL nennt man Commitment. Es gibt drei verschiedene Arten des Commitment:

- Aufgeben von Recovery-Punkten, die in RL enthalten sind (explicit commitment)
- Interaktionen zwischen einem Prozeß, der einen Recoverypunkt in RL hat, und einem, der keinen Recoverypunkt in RL hat (interaction commitment)
- Versagen des Recovery-Linien-Algorithmus (accidental commitment). Da die dafür verwendeten Routinen und Datenstrukturen sehr umfangreich werden können, sollte dieser Fall auch mitberücksichtigt werden.

Es bietet sich bei diesem Verfahren an, die Recovery-Punkte vom System selbst zu verwalten und nach Bedarf zu setzen oder aufzugeben. Sinnvoll ist ein Mechanismus dieser Art nur, wenn es effiziente Algorithmen gibt, die während des Ablaufs der Kommunikation Recovery-Punkte so definieren, daß man mit möglichst wenigen Recovery-Linien eine Fehlerbehandlung durchführen kann, die den gegebenen Anforderungen genügt. Bei einer Vielzahl von kommunizierenden Prozessen wird dieses Verfahren immer zeitraubender. Es ist schwierig, bei komplexer Interaktion überhaupt Algorithmen zu finden, die dieses Problem effizient zu lösen vermögen.

6. Das Transaktionskonzept

Für die meisten Anwendungen ist das bisher Dargelegte zu allgemein. Sie interessieren sich nur für Fehlerursachen und deren Behandlung, von denen ihre Problemlösung besonders betroffen ist. Der wirkungsvollste Schritt in diese Richtung wurde auf dem Gebiet der Datenbanksysteme gemacht (/Ver78/, /Wed76/). Die Notwendigkeit, einen umfangreichen Datenbestand in einem konsistenten Zustand zu halten, führte zum Begriff der Transaktion.

Definition 6.1:

Eine Transaktion führt einen konsistenten Datenbestand in einen neuen Zustand über, der wiederum in sich schlüssig ist. Transaktionen werden entweder vollkommen abgeschlossen oder sie verändern den Zustand des Systems nicht.

Einen Datenbestand bezeichnet man als konsistent, wenn alle Integritätsbedingungen (das sind Relationen über den Daten) erfüllt sind.

Die Konsistenz läßt sich aber nur sehr schwer feststellen, da

- die Überprüfung der Konsistenz bei großen Datenmengen sehr aufwendig werden kann,
- die Festlegung, ob die gespeicherten Informationen in sich schlüssig sind, kann unvollständig oder widersprüchlich sein,
- der Anwender selbst nicht in der Lage ist, alle Relationen anzugeben,

- mehrere Transaktionen parallel zueinander ablaufen können. Dabei nimmt der Datenbestand inkonsistente Zwischenzustände an, so daß Integritätsbedingungen nur erfüllt sein müssen, wenn es keine laufenden Transaktionen gibt.

Aus diesen Gründen ist es schwer zu prüfen, ob eine Transaktion auch wirklich keine der Integritätsbedingungen verletzt. Man geht also umgekehrt vor und sagt, daß eine erfolgreich abgeschlossene Transaktion als korrekt angesehen wird, d.h. die Daten sind in sich schlüssig. Damit liegt es zumindest zum Teil wieder in der Hand der Anwender, ihre Daten konsistent zu halten (/Reu81/). Einzelne Integritätsbedingungen lassen sich vom Datenbanksystem überprüfen.

Man führt also ein neues Strukturierungselement - die Transaktion - ein. Dabei erhebt sich die Frage, was Transaktionen von anderen Strukturierungskonzepten, wie dem Prozeßmodell (/Hof76/) und den abstrakten Objekten (/Ker82/) unterscheidet.

Aus zwei Gründen kann eine Transaktion nicht immer einem Prozeß gleichgesetzt werden:

- 1) Prozesse bleiben meist relativ lange bestehen, so daß man nicht umhin kommt, die Fehlerbehandlung feiner zu gliedern
- 2) Bei einem System kommunizierender Prozesse kann sich Recovery oft nicht auf einen Prozeß alleine beschränken.

Andererseits erscheint es nicht sinnvoll, als Einheit für die Fehlerbehandlung jedes abstrakte Objekt zu verwenden, da man sonst am Ende jeder Operation die Integritätsbedingungen überprüfen müßte. Um den Kontext im Fehlerfall wiederherstellen zu können, müssen auch die Größen einer aufrufenden Prozedur gesichert

werden. Diese Vorgehensweise würde in vielen Fällen zu einem extrem hohen Aufwand führen.

Transaktionen klammern eine Menge von Aktivitäten und geben an, wann die Operationen, die in ihrem Auftrag durchgeführt wurden, gültig werden. Transaktionen werden entweder ganz oder gar nicht durchgeführt. Weil sich parallel zueinander ablaufende Transaktionen nicht gegenseitig beeinflussen dürfen, können laufende Operationen im Fehlerfall ohne die Gefahr des Domino-Effektes rückgängig gemacht werden.

Will man die Effekte einer ganzen, richtig abgeschlossenen Transaktion im nachhinein rückgängig machen, wird dies vom System im allgemeinen nicht mehr unterstützt. Es gibt aber für viele Anwendungsfälle noch Möglichkeiten, unerwünschte Transaktionen im nachhinein aus dem System zu entfernen (z.B. Stornieren einer Buchung). Das Transaktionskonzept an sich aber bleibt einstufig, erlaubt also keine Verschachtelungen. Diese Eigenschaft beeinflußt die Anwendungen positiv wie negativ. Einerseits erleichtert das in vielen Fällen den Entwurf von Transaktionen und ermöglicht eine relativ einfache Implementierung. Andererseits aber findet der Anwender keine Unterstützung, wenn er mehrstufige Recovery-Maßnahmen benötigt. In vielen Artikeln wird betont, daß eine hierarchische Schichtung von Recoverymaßnahmen notwendig ist.

Man unterscheidet verschiedene Arten von Transaktionen:

- simple Transaktionen, die die Form "Eingabe - Verarbeitung - Ausgabe" haben,
- Dialog-Transaktionen, die während ihres Ablaufs mehrere Male mit der Umwelt kommunizieren,
- Batch-Transaktionen mit großem Umfang; sie werden meist außerhalb des normalen Dialogbetriebes oder in einer relativ abgeschlossenen Umgebung

abgewickelt,

- verteilte Transaktionen, die in mehreren Knoten eines Rechnernetzwerkes und/oder von einer Menge von Prozessen abgewickelt werden.

6.1 Synchronisation

Aus den oben angeführten Eigenschaften von Transaktionen lassen sich Bedingungen ableiten, wie die Synchronisation in solchen Systemen aussehen muß. Die Koordinierung der Zugriffe auf die Datenbank ist die unterste Schicht einer Hierarchie von Protokollen, die man für die Recovery implementiert.

Protokolle beschreiben einen Betriebsablauf, dessen Vereinbarungen von allen beteiligten Prozessen eingehalten werden müssen. Nur wenn die von ihnen festgelegten Normen beachtet werden, läßt sich ein kontrollierter Ablauf erzielen.

"Sperrprotokoll" nennt man die unterste Schicht, die für die Koordination der Transaktionen verantwortlich ist. Man versucht, so zu synchronisieren, daß es zu allen Abläufen eine serielle Abwicklung der Transaktionen gibt, die das gleiche Ergebnis liefert. Damit kann gezeigt werden, daß Transaktionen atomar sind.

Da man aber die Aufträge nicht seriell abwickeln kann oder will, führt man Sperren (locks) ein, um sich vor unerwünschten Interferenzen zu schützen. Sie verhindern, daß sich Transaktionen in unkontrollierter Weise überlappen.

Es gibt zwei Möglichkeiten, um gemeinsame Objekte zu belegen:

exclusive Die Transaktion erhält das Recht, als einzige auf eine bestimmte Menge von Sätzen zuzugreifen. Diese Sperren müssen verwendet

werden, wenn Variable modifiziert werden sollen.

shared Eine Transaktion wünscht, auf eine bestimmte Menge von Variablen zurückzugreifen, erlaubt aber anderen, diese Daten ebenfalls "shared" zu benutzen. Diese Art wird eingesetzt, um lesende Zugriffe parallelisieren zu können.

Ein weiteres Problem bei diesen Sperren ist , wie groß man die Menge der Variablen wählt, die unter ein "Lock" fallen. Wählt man diese Einheit zu groß, verringert man die Parallelität der Abläufe. Ist die Sperrgröße ("granularity of locking") zu klein, so steigt der Aufwand für das Sperren.

In /Gra79/ wird vorgeschlagen, daß die Sperren baumförmig realisiert werden. Jede Stufe untergliedert die Datenmenge weiter (z.B. Bereiche, Dateien, Satzgruppen, Sätze, Variable). Ein Eintrag in einen Knoten k des Baumes sperrt k und alle Elemente, die von k aus erreichbar sind.

In relationalen Datenbanken wird nicht nur über Adressen auf die einzelnen Sätze zugegriffen, sondern vornehmlich die zu bearbeitende Information über inhaltliche Beziehungen ausgewählt (/Esw76/). Dabei ergeben sich Probleme, wenn man sich dynamisch ändernde Beziehungen zwischen den Daten durch Sperren konsistent halten will. Dann müssen auch nicht mehr existente Objekte gesperrt oder noch nicht in den Datenbestand eingebrachte Information mit behandelt werden. Man spricht in diesem Zusammenhang auch von "Phantomsperren".

An einem Beispiel lassen sich diese Aussagen erläutern. Wir nehmen an, daß die "Datenbank" folgende Sätze enthält:

No.	Betrag
4711	20
0207	300
2404	50

Nun soll bei allen Sätzen, deren Betrag 100 übersteigt, ein Bonus aufaddiert werden. Wenn man nun parallel dazu den Betrag von No.4711 auf 120 erhöht, so ist nicht klar, ob der Bonus hier dazu addiert wird. Ähnlich verhält es sich bei der Abfrage, ob 4712 existiert und parallel dazu der Satz eingefügt wird. Transaktionen müssen also alle Tupel sperren, deren Auftreten oder Verschwinden Änderungen in den gerade bearbeiteten Relationen hervorrufen. Da es sich aber dabei um unentscheidbare Prädikate handelt, wird man sich nach anderen Lösungen umsehen müssen. Meist realisiert man dieses Problem so, daß eine Transaktion, die ein Element einer Relation ändert, diesen Zugriffspfad sperrt. Protokolle, die auf inhaltliche Beziehungen der Sätze untereinander eingehen, haben wegen ihrer Komplexität noch keine praktischen Anwendungen gefunden.

Sperren alleine erlaubt aber noch nicht, die parallelen Abläufe in eine serielle Folge abzubilden, wie es das Transaktionskonzept erfordert.

Nur wenn jede Transaktion

- vor jedem Zugriff auf eine Variable des gemeinsamen Datenbereiches diese entsprechend den Bearbeitungsanforderungen sperrt - bei lesendem Zugriff shared, bei schreibendem exclusive,
- jede Größe nur einmal sperrt,
- die Locks erst dann wieder aufgibt, wenn alle zur Durchführung eines Auftrages notwendigen Sperren gesetzt wurden (es gibt also eine Wachs-

tums- und eine Schrumpfungsphase),

dann gibt es zu jedem Ablauf eine Sequenz von Aufträgen, die - seriell durchgeführt - das gleiche Ergebnis liefern.

Diese Erkenntnis gilt als Fundamentalsatz für das Sperren in Datenbanksystemen.

6.2 Protokolle für die Verwaltung der Recovery-Daten

Die zweite Protokollschicht (/Gra79/) nach den Sperren gibt an, wie die Transaktionen ihre Informationen in den Datenbestand einbringen (DO), wie bei Fehler unvollständige Transaktionen gelöscht werden (UNDO) und wie abgeschlossene Vorgänge vor Verlust geschützt werden können (REDO).

Im einzelnen lassen sich die Aktionen des "DO-REDO-UNDO"-Protokolls aufgliedern in

DO Soll der Wert einer Variablen geändert werden, so wird dies in einem stabilen (d.h. relativ ausfallsicherem) Speichermedium hinterlegt. Dabei muß der Name der Transaktion, die Identifikation der veränderten Sätze sowie deren alter und neuer Wert vermerkt werden. Diese Aufzeichnung des Ablaufgeschehens im System wird auch Log genannt. Es enthält außer den oben angeführten Informationen noch Beginn und Ende einer Transaktion, sowie die Sicherungspunkte (Checkpoints), an denen alle bis zu diesem Zeitpunkt veränderten Sätze austransferiert wurden.

UN Soll eine Transaktion zurückgesetzt werden, weil DO ein Fehler auftrat und sie noch nicht abgeschlossen war, so werden alle Log-Einträge von ihr gesucht und die alten Werte wiederhergestellt. Diese Ak-

tionen sind in der umgekehrten chronologischen Reihenfolge des DO-Protolls durchzuführen. Nur wenn das sichergestellt ist, können die Effekte einer Transaktion aus dem Datenbestand entfernt werden.

RE Um die Ergebnisse bereits abgeschlossener Transaktionen nach Verlust (des Datenträgers bzw. weil die veränderten Daten noch nicht austransferiert wurden) wiederherzustellen, müssen alle Aktivitäten, die im Log unter ihrem Namen vermerkt sind, von vorne nach hinten aufgesucht und die Variablen auf die neuen Werte gesetzt werden.

Nach einem Systemzusammenbruch sind im allgemeinen sowohl UNDO- als auch REDO-Aktivitäten anzustoßen, je nachdem ob eine Transaktion zum Absturzzeitpunkt schon abgeschlossen war oder nicht. Dabei müssen zuerst die UNDO-Aktionen durchgeführt werden und dann das REDO-Protokoll angestoßen werden. Diese Reihenfolge ist wichtig, denn sonst könnten bereits vorher abgebrochene Transaktionen bewirken, daß Sätze auf ihren alten Wert zurückgerollt werden, obwohl sie eine nachfolgenden Transaktion wiederum verändert. Diese Restriktion braucht man nicht einhalten, wenn zurückgesetzte Transaktionen aus dem Log gelöscht werden. Das ist allerdings aus verschiedenen organisatorischen Gründen nicht zweckmäßig (alle Veränderungen des Logs sind kritisch, da es auch zu diesen Zeitpunkten zu Systemzusammenbrüchen kommen kann).

Es gibt einen fundamentalen Zusammenhang zwischen der kleinsten Sperreinheit und der Größe der Logeinträge. Man kann keine Menge von Elementen sperren, die eine Teilmenge der Daten sind, die ins Log transferiert werden (/Här80/). Sobald auf eine Log-Einheit von mehr als einer Transaktion verändernd zugegriffen werden kann, gibt es Situationen, in denen Änderungen ver-

loren gehen können.

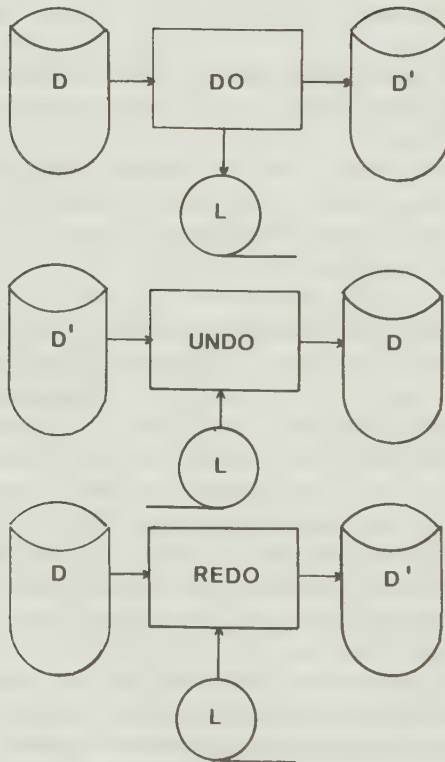


Abbildung 6.1:

DO-UNDO-REDO-Protokoll

(D ist der Zustand vor, D' der Zustand nach der DO-Aktion;

Bei der UNDO-Aktion wird das Log vom letzten Eintrag ausgehend abgearbeitet)

Nur mit Hilfe von Zusatzbedingungen läßt sich diese elementare Beziehung zwischen Sperr- und Log-Einheit auflösen (z.B. Transaktionen werden in der Reihenfolge beendet, in der sie ihre Modifikation auf die von beiden geloggtten Daten durchgeführt haben).

Logs in der hier beschriebenen Art wachsen stetig. Da damit aber auch ihr Platzbedarf und der Aufwand für die Recovery ständig steigen, sind zusätzliche Maßnahmen erforderlich, die den Umfang der Logs zu beschränken.

Eine Lösung dieses Problems wäre ein Sicherungspunkt zu einer Zeit, da keine Transaktionen mehr offen sind. Wenn alle vorher modifizierten Daten gesichert sind, so ist ein Zurückrollen von Aktionen hinter diesen Punkt nicht mehr notwendig.

Eine andere Möglichkeit besteht darin, den ältesten Eintrag einer noch nicht abgeschlossenen Transaktion und der jüngsten Protokollierung eines Sicherungspunktes zu nehmen und die UNDO- bzw. REDO-Aktivitäten von diesen Punkten aus zu starten. Bei einem Sicherungspunkt werden alle bisher gelaufenen Modifikationen so gespeichert, daß die Resultate der Änderungen auch nach Fehlern mit sehr hoher Wahrscheinlichkeit wiederzugewinnen sind.

Damit läßt sich der Platzbedarf für die Aufzeichnung des Ablaufes beschränken. Die Logs sollten auf separate Datenträger gelegt werden, um bei Ausfall eines der Massenspeicher aus einer Sicherungskopie und dem Log alle Informationen wiederzugewinnen.

Da der Verlust der Log-Daten kritisch ist, sollte man diese Daten besonders zuverlässig ablegen. Auch bei Systemzusammenbrüchen während der Log-Veränderungen muß noch definiert weitergearbeitet werden können.

Die nächsthöhere Protokollschicht regelt den Ablauf von Datenmanipulation und Log-Eintrag.

Nach dem WAL-Prinzip (WAL = write-ahead-log) darf man

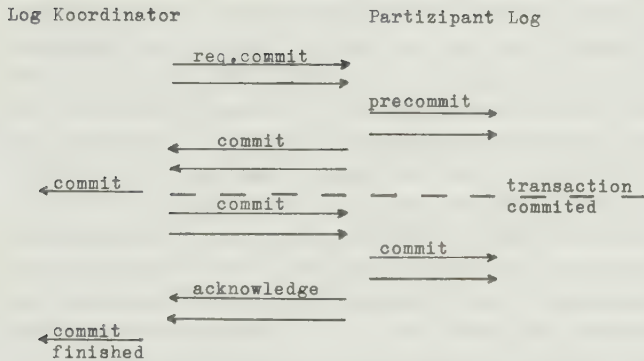
eine Änderung erst dann durchführen, wenn sie vorher im Log aufgezeichnet wurde. Kann eine Modifikation dann nicht mehr realisiert werden, so wird nur von der UNDO-Prozedur die noch nicht geänderte Variable mit ihrem alten Wert überschrieben. Der Log-Eintrag, daß eine Transaktion erfolgreich abgeschlossen worden ist, muß vor Freigabe der Sperren erfolgen.

6.3 Abschluß einer Transaktion

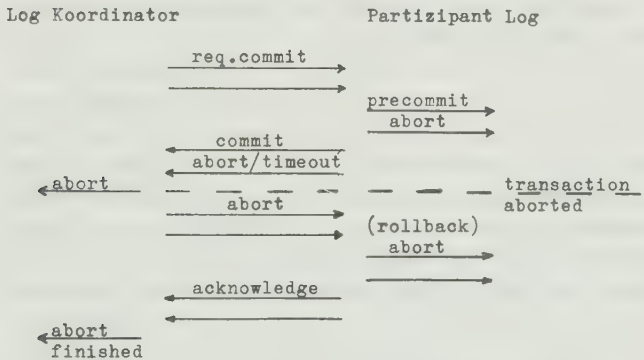
Den erfolgreichen Abschluß einer Transaktion bezeichnet man als Commitment, den Abbruch als Abort. Die Endebehandlung läßt sich relativ leicht durchführen, wenn nur ein Prozeß betroffen ist. Der Fall kompliziert sich, wenn mehrere Prozesse eine Transaktion realisieren. Dann kann die einzelne Komponente nicht mehr entscheiden, ob eine Transaktion korrekt ausgeführt worden ist oder nicht. Es ist ein Koordinator notwendig, der dies entscheidet und den Log-Eintrag vornimmt.

Das Commit-Protokoll läuft dann wie folgt ab:

- 1) alle beteiligten Prozesse teilen dem Koordinator mit, ob ihre Aktionen erfolgreich abgeschlossen werden konnten. Dieses Ergebnis wird auch in der Aufzeichnung des Ablaufgeschehens hinterlegt.
- 2) Antworten alle positiv, so schreibt der Koordinator "committed" ins Log und teilt allen beteiligten Prozessen den erfolgreichen Abschluß mit. In allen anderen Fällen sendet der Koordinator "abort" an alle Komponenten und hinterlegt dieses Ergebnis auch im Log.
- 3) Alle beteiligten Prozesse und der Koordinator vermerken sich in ihrem eigenen Log den positiven oder negativen Abschluß der Transaktion.



erfolgreiche Transaktion



abgebrochene Transaktion

Abbildung 6.2:

Two-Phase-Commitment-Protocol

Dieses Ablaufschema heit "Two-Phase-Commitment-Protocol" (/Gra79/).

Probleme ergeben sich bei diesem Verfahren, wenn die ausgetauschten Nachrichten beim Transport verloren gehen. Dann können Fehler auftreten, die mit diesem einfachen Protokoll nicht mehr behebbar sind. Es läßt sich zeigen, daß es bei Fehlern dieser Art kein endliches Protokoll gibt, das dieses Problem löst ("Generals-Paradox"). Beispiele in diesem Fall sind der Verlust von "Commit"-Nachrichten der Prozesse oder des Koordinators. Es ist jedoch möglich, die Verlustwahrscheinlichkeit beliebig klein zu halten.

Zwischen dem Abschließen der lokalen Aktivitäten und dem "commit/abort"-Nachricht des Koordinators müssen die Änderungen sowohl rücksetzbar (UNDO) als auch wiederherstellbar (REDO) sein, da das Schicksal der Transaktion zwischen diesen beiden Ereignissen vom Koordinator entschieden wird.

Dafür benötigt man zusätzliche Unterstützung durch die lokalen Log-Komponenten.

Wenn alle Transaktionen diese Protokolle beachten und korrekt durchführen, ist trotz Rechnerausfall, Zerstörung eines Datenträgers oder abgebrochener Transaktionen mit hoher Wahrscheinlichkeit auszuschließen, daß Änderungen durch erfolgreich abgeschlossene Transaktionen für immer verloren gehen.

Kritisch ist bei diesem Konzept neben dem Einhalten der Protokolle der Verlust von Log-Einträgen.

7. Modellierung von parallelen Berechnungen mit Fehlerbehandlung

Um die Bedeutung von Aussagen veranschaulichen zu können, ist eine fundierte Basis notwendig. Deshalb soll an dieser Stelle auf die im ersten Kapitel formulierten Grundlagen zurückgegriffen werden. Zwar ist es möglich, auf bestehende Modelle aufzubauen, doch wäre ein Rückgriff auf die verschiedenen Begriffswelten nur schwer nachzuvollziehen und wenig anschaulich. Daß es trotzdem viele Querbezüge zu einer ganzen Reihe von Konzepten und Meinungen gibt, ist unvermeidbar und bestimmt auch wünschenswert.

Die Rückbesinnung auf die Grundlagen hat auch den Vorteil, daß man einzelne Problemstellungen besser in ein Gesamtkonzept einordnen kann und so besser den Überblick behält.

7.1 Zielvorstellungen

Als Basis wollen wir das Prozeßmodell betrachten, das in der Begriffsdefinition bereits ausführlich erläutert wurde.

Auf eine hierarchische Gliederung der Systeme sei zumindest bei den ersten Überlegungen verzichtet. Als "Fehler" wird hier die Abweichung des Systems von seinem spezifizierten Verhalten angesehen. Im Prozeßmodell existieren also noch zusätzliche, nicht im Entwurf vorgesehene Übergänge und Zustände. Auch die Recovery selbst bewirkt Änderungen im System, die im normalen Ablauf nicht wirksam werden.

Um die zu behandelnden Lösungen besser beurteilen zu

können, sollen die Ziele der Recovery an einem vereinfachten Modell etwas präziser dargestellt werden.

Die dabei gewonnenen Vorstellungen sind nützlich als Maßstab für die Leistungsfähigkeit konkreter Recovery-Methoden.

Das Ziel der Fehlerbehandlung ist die Rückkehr von fehlerhaften Zuständen zu den normalen Berechnungspfaden.

Definition 7.1:

Ein F-R-System (Fehler-Recovery-System) besteht aus einer Menge von Zuständen (E) und Zustandsübergängen (B).

B ist eine Teilmenge von ExE .

Die Menge $K \subseteq E$ beinhalte die konsistenten Zustände, wie sie nach dem spezifizierten Verhalten auftreten.

$I \subseteq E$ umfaßt die inkonsistenten Zustände.

K und I sind disjunkt.

Alle Übergänge, die aus K in I führen, gehören zur Fehlerrelation F.

R stelle eine Beziehung zwischen I und K her (Recoveryrelation).

Alle Kanten des Graphen werden durch eine Funktion b bewertet. b bildet die Elemente aus B in O (Operationen) ab, wobei es in der Menge O ein nil-Element gibt, das nur Knoten verbinden kann, deren Quell- oder Zielzustand in I liegen.

Als aktive Fehlerübergänge kann man diejenigen Defekte bezeichnen, die durch fehlerhafte Ausführung einer Operation entstehen. In diesem Fall fehlt eine Kante zwischen zwei konsistenten Zuständen, dafür gibt es eine Verbindung mehr zwischen K und I.

Passive Fehlerübergänge entstehen ohne Einwirkung von Prozessen und sind auf Fehlverhalten von Informationsträgern zurückzuführen. Durch passive Fehlerübergänge

entstehen zusätzliche Kanten zwischen K und I mit Bewertung nil.

Definition 7.2:

Ein Fehler fa (aktiver Fehlerübergang) ist ein Paar (k,i) , dem ein (k,l) des korrekten Graphen zugeordnet ist ($b(k,l)=b(k,i)$). Dabei sind k und l Elemente von K und i aus I . Die Menge aller aktiven Fehlerübergänge sei Fa .

Passive Fehlerübergänge sind alle Elemente $fp=(k,i)$ aus Fp , der Restmenge von $F - Fa$. Es muß gelten $b(k,i)=\underline{nil}$.

Definition 7.3:

Eine Recovery-Relation bezeichnet man als optimal (R_{opt}), wenn gilt:

- Für alle Elemente i aus I , zu denen ein Pfad von einem k aus K führt, gibt es einen Pfad zu einem k' aus K .
- Wenn (i,k') aus R_{opt} , und es Pfade von k nach i und k' gibt, so ist die Bewertung eines Pfades von k nach k' (wobei alle Elemente aus $K \times K$ sind) gleich der Bewertung des Pfades von k nach i und von i nach k' (nil-Elemente bleiben dabei unberücksichtigt).

Diese Tatsache, daß zwei Pfade zu demselben Zielzustand führen, ist auch als Konfluenzeigenschaft bekannt (/Ker74/).

Bei den Definitionen 7.1 - 7.3 bleibt das Ein-/Ausgabeverhalten eines Systems unberücksichtigt.

Als nützlich erweisen sich diese Ideale als Gütekriterien für verwirklichte Lösungen. Man wird also nach einem R suchen, das bei den gegebenen Randbedingungen dem R_{opt} möglichst nahe kommt.

Wie gut diese Zielvorstellung für die Recovery genutzt werden kann, hängt aber stark von der Art des Fehlers ab.

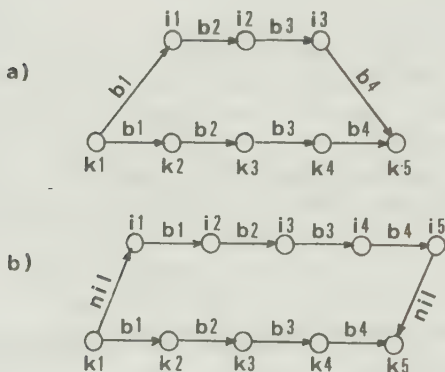


Abbildung 7.1:

F-R-System mit Ropt

a) bei aktivem Fehlerübergang

b) bei passivem Fehlerübergang

Eine Ropt entsprechende Fehlerbehandlung ist zu realisieren, wenn der Fehlertyp mit seinen Auswirkungen bekannt ist und man genügend Redundanz zur Verfügung hat. So lassen sich manche Fehler in der Verkettung von Datenstrukturen dadurch erkennen und beheben, daß man zusätzliche Verknüpfungen realisiert. Falsche Aktionen in der Verwaltung der Informationen können dabei erkannt und beseitigt werden (/Tay80a,b/).

Auch Störungen auf den Übertragungswegen bei Rechner-Rechner-Kommunikation kann man auf ähnliche Art und Weise behandeln. Durch Wiederholungsanforderung versucht man, die Störungen zu beheben.

Wenn aber die Fehlerursache nicht ganz genau zu analysieren ist, so kann man Ropt nicht so ohne weiteres

verwenden. Dasselbe gilt, wenn die Auswirkungen nur ganz grob zu kalkulieren sind. In diesen Fällen besteht die einzige Chance zur Reparatur darin, auf einen konsistenten Zustand zurückzugehen und von da aus einen alternativen Berechnungspfad einzuschlagen.

Wie man sich das vorstellen könnte, ist bereits im Zusammenhang mit dem Recovery-Block-Konzept erläutert worden. Für die weiteren Betrachtungen ist es nützlich, wenn die Akzeptanz einer Operation sowohl im Anschluß an diese Operation (wie beim Recovery-Block-Konzept) als auch von anderen Aktivitäten überprüft werden kann. Es soll also bei den folgenden Überlegungen auch möglich sein, daß die Überprüfung der Aktivitäten an beliebiger Stelle erfolgen kann. Damit trägt man den Anforderungen von Prozeßsystemen besser Rechnung, in denen die erzeugende Einheit die Richtigkeit der Daten nicht vollständig prüfen kann (siehe Abbildung 7.2).

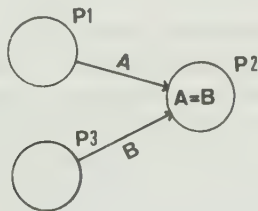


Abbildung 7.2:

Akzeptanztest in einem dritten Prozeß

Es ist klar, daß damit ein Strukturierungselement, der zentralisierte Akzeptanztest, wegfällt. Man wird also versuchen müssen, Akzeptanztests auf andere Art und Weise in das Konzept einzuordnen.

Die alternativen Implementierungen zur Behandlung von

Entwurfsfehlern können beibehalten werden. Ein solches Konstrukt wird dann als Recovery-Gruppe bezeichnet. Die für die Recovery notwendige Redundanz ist auf zwei Arten zu realisieren: in die normalen Daten integriert oder separat in eigens dafür geschaffene Segmente. Die getrennte Speicherung redundanter Daten bietet besseren Schutz und leichtere Modifizierbarkeit. Eine Integration dieser Redundanz ist oft erwünscht, um die Effizienz des Systems zu erhöhen.

7.2 Ein Recovery-Modell

Die Aktionen, die insgesamt die Aktivitäten eines Systems darstellen, sind nicht unabhängig voneinander. Da gibt es Operationen, die in einer festgelegten Reihenfolge abgewickelt werden müssen, wie innerhalb eines sequentiellen Prozesses. Nachrichten werden ausgetauscht und geben dem Absender die Möglichkeit, Informationen, aber auch Fehler weiterzuleiten. Die Transitionen in einer großen Datenbasis sind im allgemeinen abhängig von den vorausgegangenen Veränderungen.

Diese Verflechtung von Aktionen macht die Fehlerbehandlung erst problematisch. Im allgemeinen sind alle Aktivitäten miteinander verknüpft durch die Transitivität der Abhängigkeit. Es muß also weitere Beziehungen geben, die das Ausmaß der Recovery beschränken. Nur solche Aktivitäten, die den Defekt verursacht haben können oder davon betroffen sind, dürfen zurückgesetzt werden. Dazu ist eine Relation nötig mit der Aufgabe, die aktuelle Berechnung in zwei Teile zu zerlegen:

- die von einer Recoverymaßnahme betroffenen
- und
- die Aktivitäten, die nicht mehr rückgängig gemacht werden müssen.

Diese Relation zur Beschänkung der Recovery wird unter der Bezeichnung "Validierungsrelation" (VAL) in diesem Abschnitt eingeführt. Zusammen mit der "Abhängigkeitsrelation" (INF) bestimmt sie das Recovery-Verhalten des Modells.

Dabei existieren Abhängigkeiten dieser Relationen untereinander.

Im folgenden sei A eine Menge von atomaren Aktionen, s sei der momentane Zustand des Systems.

$\text{dom}(s) = \text{dom}(v_1) \times \dots \times \text{dom}(v_n)$

$\triangleright$ ist eine Halbordnung über A. $a_1 \triangleright a_2$ bedeutet, daß im Rahmen einer konkreten Berechnung Aktion a_1 vor a_2 ausgeführt wurde.

s_t^v drückt aus, daß im Zustand s der Variablen v der Wert t zugewiesen wird.

$a(s)$ ist der Zustand von s nach Ausführung von a.

Die Prädikate ändert und fragt ab sind in /Ker82/ definiert:

a ändert v, wenn v nach a einen anderen Wert besitzt als vorher.

a fragt ab v, wenn

a ändert v und ein t aus $\text{dom}(v)$ existiert, mit $a(s) = a(s_t^v)$.

oder

a ändert v nicht und ein t aus $\text{dom}(v)$ existiert, mit $(a(s)_t^v = a(s_t^v))$

gilt.

Definition 7.4:

Aktion a_1 beeinflusst a_2 über die Variable v im weiteren Sinne ($\text{INF}(a_1, a_2, v)$), wenn gilt:

1. a_1 wird zeitlich vor a_2 abgewickelt

($a_1 \triangleright a_2$)

2. a_1 ändert v und a_2 fragt ab v

3. $\neg \exists a_3 \in A(a_1 \triangleright a_3 \triangleright a_2 \rightarrow (a_3 \text{ ändert } v))$

Dabei wird davon ausgegangen, daß $a_j \triangleright a_k$ oder $a_k \triangleright a_j$ gilt, wenn a_j ändert und a_k ändert oder fragt ab. Dies zu garantieren, ist eine der vielen Aufgaben, die der Synchronisation obliegt. Wie stark sich Koordinierungsfunktionen und Recoverymechanismen gegenseitig beeinflussen, zeigt sich wie hier an vielen anderen Stellen auch. Ohne klar ausgeprägte Synchronisation ist keine Fehlerbehandlung sinnvoll möglich, andererseits entstehen durch die Prozeßkommunikation Probleme, da sich dabei auch Defekte ausbreiten.

Warum Definition 7.4 für manche Anwendungsfälle zu restriktiv ist, läßt sich mit einem einfachen Beispiel belegen (vor a_1 ist der Wert der Variablen v 5):

(a1) $v := 3;$

(a2) if $v = 2$ then $x := 1$ else $x := 2;$

Es ist klar, daß $\text{INF}(a_1, a_2, v)$ nach Definition 7.4. Andererseits aber kann a_1 in dieser Konstellation das Ergebnis von a_2 nicht beeinflussen.

Eine Operation a_2 hängt im engeren Sinne von einer anderen Aktion a_0 ab, wenn durch a_0 Wahrheitswerte von Bedingungen verändert werden. Mit dieser Erkenntnis läßt sich die Abhängigkeit im engeren Sinne formulieren.

Definition 7.5:

Eine Aktion a_1 beeinflußt im engeren Sinne die Operation a_2 über die Variable v ($\text{inf}(a_1, a_2, v)$), wenn gilt:

1. $a_1 \triangleright a_2$

2. a_1 ändert v von t nach t'

und a_2 fragt ab v

3. wenn a_2 ändert v gilt, dann ist $a_2(s_t^v) = a_2(s)$
 wenn a_2 ändert v nicht, dann ist $a_2(s_t^v) = a_2(s)_t^v$

4. $\neg \exists a_3 \in A(a_1 > a_3 > a_2 \text{ und } \inf(a_3, a_2, v))$

Meist spielt es keine Rolle für die folgenden Untersuchungen, ob die weitere oder engere Form von Abhängigkeit verwendet wird. Nur im Recoveryfall kann es Probleme geben. Während man bei INF bedenkenlos den Wert von v rücksetzen kann, muß bei der strengeren Version bedacht werden, daß es zwischen a_1 und a_2 eine weitere Aktion a_3 geben kann, die den Wert dieser Variablen ändert, ohne a_2 zu beeinflussen. In diesem Fall liefert dann die Restauration des alten Wertes einen falschen Zustand.

Im allgemeinen kann man sagen, daß durch die strenge Abhängigkeit wesentlich zutreffender die Ursache von Defekten herauszufinden ist. Dafür ist aber eine Recovery erheblich aufwendiger.

Problematisch werden die Definitionen 7.4 und 7.5, wenn im Zuge eines Fehlers falsche Abhängigkeiten entstehen.

Um die Recovery mit hoher Wahrscheinlichkeit erfolgreich durchführen zu können, versucht man, Abhängigkeiten möglichst gut zu erfassen. Die Beeinflußt-Relationen sollten deshalb aus einer Spezifikation entnommen werden. Unabhängig von den anderen Implementierungen sind die Abhängigkeiten festzuhalten. Im laufenden Betrieb kann man dann die konkreten Abhängigkeiten überprüfen und bei Fehlern sofort Recovery-Maßnahmen einleiten.

Mit einem Vergleich von spezifiziertem INF/inf und implementiertem Verhalten lassen sich die besten Er-

gebnisse erzielen. Die Überprüfung kann ähnlich organisiert werden wie dies bei Schutzkonzepten (/Lin78/) zu realisieren ist.

Die Abhängigkeiten darf man nicht zu komplex darstellen, damit nicht durch ihre Definition weitere Fehlerquellen entstehen. Die Relationen automatisch zu ermitteln, ist nur in Spezialfällen möglich.

Aus diesem Grunde ist ein modularer Aufbau des Gesamtsystems nicht nur wünschenswert, um Fehler zu vermeiden, sondern auch zur Fehlereindämmung und zur Ermittlung des Umfangs der Recovery. Auch andere Modelle gehen von der Prämisse aus, daß im Großen und Ganzen bekannt ist, wie sich einzelne Operationen gegenseitig beeinflussen (/Kan81/, /Ran79/ und /Rus80/).

Die dort angeführten Beispiele zeigen deutlich, daß die Schwierigkeiten beim Übergang von Spezifikation und Implementierung lösbar sind. Wichtig erscheint dabei, sich dieses Problems bewußt zu sein, wenn man die INF- oder inf-Relation definiert.

Die Abhängigkeiten zwischen Modifikationen und Abfragen einer Variablen werden oft mit dem Read-Write-Modell dargestellt (/Wed76/, /Esw76/). Danach gibt es unerwünschte Interferenzen zwischen zwei Operationen nicht, wenn sie nur lesend auf gemeinsame Variable zugreifen. Dies zu gewährleisten, ist eine wichtige Aufgabe der Synchronisation (/Ker82/). Es erscheint allerdings nicht sinnvoll, auch bei der Fehlerbehandlung von dieser Anschauung auszugehen. Da Konflikte beim Zugriff auf gemeinsame Variable bereits bei der Durchführung der Aktion behandelt werden (siehe dazu auch die Bemerkungen zu Definition 7.4), spielt hier nur noch der Informationsfluß zwischen den einzelnen Aktivitäten eine Rolle. Dies wird durch die Abhängigkeit der Abfragen von vorangehenden Änderungen ausgedrückt.

Durch das Read-Write-Modell können Bedingungen vorge-

geben werden, die dann in einem konkreten Ablauf zu einer INF-Beziehung führen.

Ein Problem der fragt ab-Relation sollte nicht unerwähnt bleiben:

(a1) v1:=true

(a2) v2:=true

(a3) if v1 or v2 then ...

Im engeren Sinne von "fragt ab" besteht keine Beziehung zwischen a1, a2 und a3. Fehler sowohl in a1 als auch in a2 könnten das Resultat verfälschen. Defekte dieser Art zu analysieren, kann sehr problematisch sein, da sie nur unter extremen Randbedingungen auftreten.

Doch auch hier bietet sich eine Lösung an, wenn man die Verbundvariable $v3 = (v1, v2)$ einführt.

Auf diese Art und Weise lassen sich auch noch kompliziertere Zusammenhänge modellieren.

An dieser Stelle sollte der Begriff "Variable" noch etwas näher erläutert werden.

Eine Variable besitzt einen Namen und einen Wert, der sich im Laufe einer Berechnung ändert. Variable können auch erzeugt und wieder zerstört werden. Damit ist aber noch nichts darüber ausgesagt, welchen Wertebereich eine einzelne Größe besitzt. Wie der Grunddatenbereich in Variable zerlegt ist, bestimmt die Parallelität der einzelnen Operationen. Darüber hinaus beeinflusst diese Gliederung den Aufwand für die Recovery und die Strukturen der INF-/inf-Relationen. Im folgenden wollen wir nur "INF" schreiben, und die konkrete Realisierung dabei außer acht lassen.

Damit aber die Fehlerbehandlung überhaupt sinnvoll beschränkt werden kann, ist die Validierungsrelation

VAL (Bestätigung der Richtigkeit) notwendig.

Definition 7.6:

VAL (a_1, a_2, v) ist ein dreistelliges Prädikat über zwei Aktionen und einer Variablen.

a_1 validiert die Änderungen der Variablen v durch a_2 , wenn:

1. a_1 mit Hilfe einer Überprüfungsroutine feststellt, daß die Änderung von v durch a_2 akzeptiert werden kann.
2. Die Variable v kann auch indirekt von a_1 abgefragt werden. Es muß aber einen Informationsfluß von a_2 nach a_1 über v geben, das heißt, es existieren Aktionen $ak_0, \dots, ak_n$ aus A mit $INF(ak_{i-1}, ak_i, v_i)$, $i=1, \dots, n$; $ak_0=a_2$, $ak_n=a_1$, $v_1=v$ und mit $j=i-1$ existieren t aus $\text{dom}(v_j)$ für die gilt $\text{value}(v_i, ak_i(s_t^{v_j})) = \text{value}(v_i, ak_i(s))$ ($\text{value}(v_i, s)$ liefert den Wert der Variablen v_i im Zustand s).

Eine Aktion wird als validiert angesehen, wenn sämtliche Änderungen validiert worden sind.

Bei relativ simplen Datenstrukturen lassen sich auch einfache Validierungsfunktionen finden. Schwierig ist es, umfangreiche, komplexe Größen zu prüfen. Deshalb ist auch hier die Forderung angebracht, daß Datenstrukturen in möglichst übersichtliche Teile zu zerlegen sind.

Mit Hilfe von VAL läßt sich der Aufwand für die Recovery beschränken. Existieren sowohl VAL(a_3, a_1, v) als auch INF(a_1, a_2, v), so braucht bei Defekten, die a_2 in Mitleidenschaft ziehen, a_1 nicht mehr rückgesetzt werden. Ist a_1 noch nicht validiert, so ist auch diese Aktion bei der Fehlerbehandlung mit zu berücksichtigen.

Zur übersichtlichen Darstellung der Zusammenhänge ist eine graphische Darstellung nützlich.

Definition 7.7:

Ein Recovery- oder INVA-Graph ist eine Überlagerung der Strukturen $G1=(N1,E1,b1)$ und $G2=(N2,E2,b2)$, mit

$N1=N2=A$,
 $E1,E2$ sind Teilmengen von $A \times A$,
 V : Variablenmenge,
 $b1: E1 \rightarrow \mathcal{P}(V)$, $b2: E2 \rightarrow \mathcal{P}(V)$,
 $e=(a1,a2)$ in $E1 \rightarrow \text{INF}(a1,a2,v)$, $v \in b1(e)$
 $e=(a1,a2)$ in $E2 \rightarrow \text{VAL}(a1,a2,v)$, $v \in b2(e)$

Nach diesem Graphen sollen die hier formulierten Zusammenhänge als INVA-Modell bezeichnet werden. Zur übersichtlicheren Darstellung werden die beiden Teilgraphen in einem Bild zusammengefaßt. Dabei sind die VAL-Kanten durch gestrichelte Linien repräsentiert.

Die zu einer Recovery-Gruppe gehörenden Aktionen werden im Bild durch einen Kreis um die relevanten Knoten dargestellt.

7.3 Fehlerausbreitung

Durch eine Berechnung in diesem System wird nun der Zustand s verändert. Es werden neue Recovery-Gruppen definiert und weitere Knoten und Kanten in den INVA-Graphen eingeführt. Diese Veränderungen stellen die Vorwärts-Aktivitäten des Systems dar.

Unter der Menge von neuen Knoten kann es auch welche geben, die sich nicht erfolgreich abschließen lassen.

- (R2) Wird eine Aktivität a_1 zurückgesetzt, so sind alle a_2 aus A mit $\text{INF}(a_1, a_2, v)$ von der Recovery betroffen.
- (R3) Wenn ein Element a einer Recovery-Gruppe A_1 zurückgesetzt wird, so auch alle anderen Elemente von A_1 . Voraussetzung ist, daß es keine Menge A_2 gibt, die in A_1 enthalten ist und der a auch angehört.

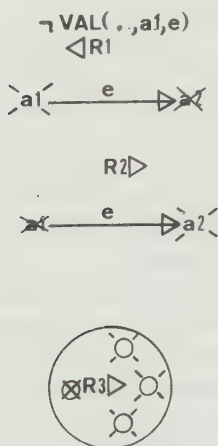


Abbildung 7.4:
 Recovery-Regeln

Die Begründung der Regeln 2 und 3 liegt auf der Hand. Die Einschränkung von R3 dient nur dem Zweck, daß bei geschachtelten Recovery-Gruppen nur der innerste von der Fehlerbehandlung betroffen ist. Wer eine Recovery-Gruppe definiert, will damit aussagen, daß entweder alle Aktivitäten positiv abgeschlossen werden müssen

oder keine. Wie bei einer Transaktion gibt es also keine "fast richtigen" Recovery-Gruppen. Nur wenn die Gruppen ineinander geschachtelt sind, verlieren die äußeren während der Bearbeitung der innersten Gruppe ihre Gültigkeit bezüglich R3.

Wenn eine Änderung ungeschehen gemacht wird, so beeinflusst das auch diejenigen Aktionen, die auf dieser Modifikation aufbauen (R2).

Die Notwendigkeit von R1 ist nicht so offensichtlich wie die der beiden anderen Regeln. Wie bereits ausgeführt, kann ein Fehler nicht nur durch interne falsche Berechnungen zustande kommen, sondern auch durch fehlerhafte Eingangsparameter. Wenn aber die Änderungen bereits validiert sind, braucht die Recovery nicht auf die ändernde Aktion ausgebreitet zu werden.

Betrachtet man die drei Regeln, so scheint R2 unkritisch bezogen auf den Domino-Effekt zu sein. Für sich alleine gesehen, birgt sie ja auch keine Gefahren in sich, weil sie nur zum Rücksetzen von Folgeaktivitäten führt, die es ja nach einem entdeckten Fehler im ersten Schritt gar nicht gibt. Doch im Zusammenhang mit den beiden anderen Regeln kann sie zu einem großen Problem werden, da durch sie auch bereits validierte Operationenfolgen von der Recovery in Mitleidenschaft gezogen werden.

Regel R3 dehnt die Recovery nicht in eine bestimmte Richtung aus, sondern über eine "Fläche". Auch hier gilt, daß die Recovery leicht zu beschränken wäre, gäbe es alleine nur diese Regel. Dann wären nur die Recovery-Gruppen betroffen, die mindestens eine fehlerhafte Aktion enthielten. In Verbindung mit R1 und/oder R2 treten aber auch hier Domino-Effekte auf.

Obwohl mit der Anwendung von R1 alleine ein nicht beschränktes Rücksetzen möglich ist, treten die realen Probleme erst bei den Kombinationen der verschiedenen Recovery-Regeln auf.

Die Anwendung der Regeln läßt sich weiter strukturieren. Da R1 nur auf AI und auf Aktionen angewandt werden kann, die man durch R1 erreicht, ist es möglich, die Anwendung dieser Regel vorzuziehen. Damit wird eine Menge INFL erfaßt. Danach können R2 und R3 angewandt werden und zwar R2 ausgehend von INFL, Regel R3 muß auch AI miteinbeziehen (siehe dazu auch Abbildung 7.5).

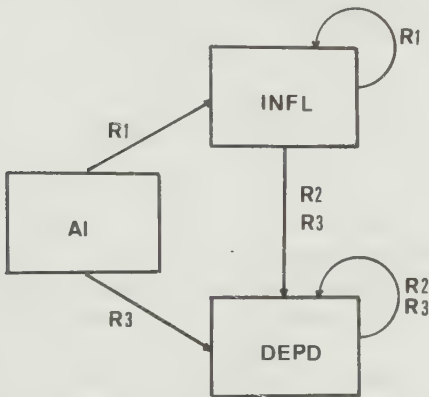


Abbildung 7.5:
Strukturierung der Recovery-Regeln

INF, VAL und die Recovery-Regeln können natürlich auch auf sequentielle Programme angewandt werden.

Validierungen müssen während der Recovery gültig bleiben, auch wenn die validierende Aktion von der R1-Recovery-Regel erfaßt wird. Diese Anforderung muß erfüllt sein, sonst würde eine Recovery bei sequentiellen Abhängigkeiten nicht zu beschränken sein. Ist die Anwendung der Recovery-Regel erschöpft, können die Validierungen rückgängig gemacht werden, die von Aktionen ausgehen, die von der Recovery betroffen waren. Damit ist auch die Behandlung falscher Validierungen möglich, jedoch ist dazu ein weiterer Recovery-Schritt

erforderlich.

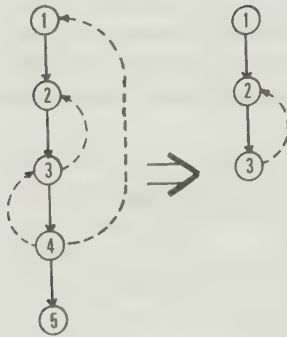


Abbildung 7.6:

Sequentielle Abhängigkeiten

(die Validierungen von 4 aus begrenzten die

Recovery, werden aber dann gestrichen)

Manche Modelle (/Shr81/) unterscheiden bei der Fehlerbehandlung zwischen Softwaredefekten (error recoverability) und Hardwareausfällen (crash resistance). Diese Verfeinerung könnte sich hier so auswirken, daß man die R1-Propagation nur im Falle einer "negativen" Validierung (Nichtakzeptanz im Sinne des Recovery-Block-Konzepts) zuläßt.

Kann man davon ausgehen, daß bei einem Ausfall nur die Aktionen aus AI Defekte enthalten, so ist die Anwendung von R1 nicht erforderlich.

Es wurde in diesem Abschnitt noch nichts darüber ausgesagt, wie sich Aktionen rückgängig machen lassen. Probleme treten auf, da Aktivitäten im allgemeinen nicht in der umgekehrten Reihenfolge zum normalen Ablauf von der Recovery erreicht werden. Damit ist das UNDO-Protokoll, wie es beim Transaktionskonzept ein-

gesetzt wird, nicht direkt anwendbar. Man muß vermerken, in welcher Reihenfolge Änderungen durchgeführt wurden.

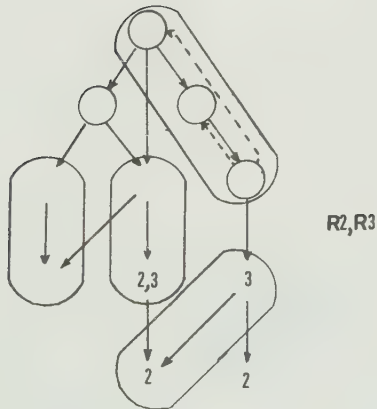
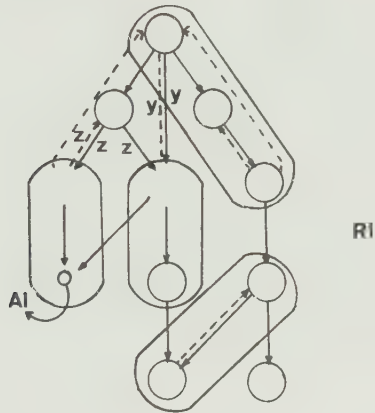


Abbildung 7.7:
Anwendung der Recovery-Regeln auf den
INVA-Graphen von Abbildung 7.3

Das Rücksetzen einer Modifikation der Variablen v durch Aktion a ist nur dann erforderlich, wenn der augenblickliche Wert "jünger" ist als die Änderung von v durch a . Dies kann man durch eine fortlaufende Nummerierung der Änderungen erkennen.

Bevor weitere Untersuchungen angestellt werden, ist zu erörtern, ob das Modell minimal ist. Diese Eigenschaft hat es dann, wenn es keine Elemente und Regeln gibt, die auf die restlichen Darstellungsformen abgebildet werden können.

Die Regeln $R1$ und $R2$, genauso wie INF und VAL beschreiben so unterschiedliche Sachverhalte, daß eine Transformation des einen auf das andere unvorstellbar erscheint.

Anders verhält es sich bei Recovery-Gruppen und der Regel $R3$.

Offensichtlich ist es möglich, ein durch $R1$ angestoßenes Rücksetzen von Recovery-Gruppen zu simulieren. Dabei braucht man nur je Recovery-Gruppe $A1$ einen zusätzlichen Zustand a' einzuführen und zu jedem a_i aus $A1$ eine Kante ziehen. Damit erfüllt $A1$ die Eigenschaft, daß bei einer Ausbreitung der Fehlerbehandlung auf $A1$ durch $R1$ -Propagation alle a_i von der Recovery betroffen sind.



Abbildung 7.8:

Simulation von $R3$ bei Anstoß über $R1$

Durch R2-Propagation verursachtes Rücksetzen ist nicht so ohne weiteres zu simulieren (siehe dazu auch Abbildung 7.9). Wird über a_3 durch R2 Aktion a_2 zurückgesetzt, so folgt aus R3, daß auch a_1 als ungültig zu erklären ist. Aus den vorangehenden Aussagen über die hierarchische Anordnung der Recovery-Regeln folgt, daß zur Simulation von R3 in diesem Fall nur mehr R3 angewandt werden darf. Also gibt es einen Pfad von a_2 nach a_1 , um a_1 rückzusetzen. Da es aber einen Pfad von a_1 über a_3 nach a_2 gibt, entsteht ein Widerspruch zu der Anforderung, daß die INF-Relation zyklensfrei sein muß. Wenn nun keine weiteren Einschränkungen der INF-Relation vorgenommen werden soll, so ist offensichtlich, daß dabei ein Zyklus entstehen muß, der durch a_1 , a_3 und a_2 führt.

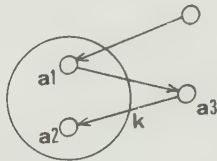


Abbildung 7.9:
Recovery-Zyklen

Auch die "normale" R2-/R3-Recovery erreicht in diesem Falle zweimal a_3 . Es ist klar, daß eine Simulation dieses Verhalten widerspiegeln muß. Damit wird deutlich, welche Rolle die Recovery-Gruppen unter dieser Betrachtungsweise spielen. Sie führen bei nichtatomaren Recovery-Gruppen Zyklen in der Fehlerbehandlung ein.

Durch R2 und R3 entstehen Zyklen genau dann, wenn es nichtatomare Recovery-Gruppen gibt. Mit der in Kapitel 1 dargelegten Definition läßt sich diese Behauptung

tung direkt belegen.

Zyklen sind so definiert, das es Knoten a_i gibt, die wieder durch R2- oder R3-Propagation erreicht werden, wenn die Recovery von a_i ausgeht. Sie lassen sich beim Durchführen der Recovery aber relativ leicht erkennen. Nach der Identifikation eines Zustands, der schon einmal erreicht wurde, kann dieser Weg für die Fehlerbehandlung als abgeschlossen gelten.

Wenn im folgenden von Pfaden die Rede ist, wollen wir voraussetzen, daß Schleifen in einem Pfad höchstens einmal durchlaufen werden.

Wenn man durch R2 angestoßenes Rücksetzen eines Recovery-Blocks A1 simulieren will, müssen folgende Ergänzungen vorgenommen werden.

- 1) Einen Zustand a'' , der sich selbst validiert
- 2) Kanten von a'' zu allen Zuständen aus A1
- 3) Wenn es eine Kante gibt zwischen einem a_k , das nicht zu A1 gehört, zu einem a_l aus A1, so gibt es auch eine Verbindung zwischen a_k und a'' .

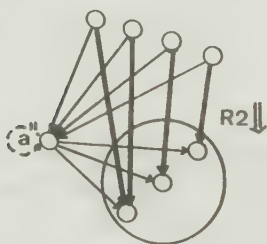


Abbildung 7.10:

Simulation von R3 bei Anstoß über R2

Der Knoten a'' ist immer validiert, a' nie. a'' und a' sind notwendig, da bei $a' = a''$ die R1-Regel nicht wirk-

sam wird.

Das Rücksetzverhalten braucht wegen Zyklen nicht weiter modifiziert zu werden. So können sich die folgenden Aussagen über den Domino-Effekt nur auf Systeme ohne Recovery-Blöcke beziehen, die aber Zyklen in der INF-Relation enthalten dürfen.

7.4 Der Domino-Effekt

Wenn man über den Aufwand für Recovery redet, so muß zuerst einmal geklärt werden, über welche Belastungen des Systems man eigentlich spricht.

Es gibt drei Arten von Beeinträchtigungen, die durch die Fehlerbehandlung entstehen:

- 1) Belastung des normalen Systemablaufs durch die Vorsorge für die Recovery
- 2) Aufwand für die Recovery selbst
- 3) Verlust an Berechnungsschritten durch Rücksetzen

Die Belastung im normalen Betrieb ist proportional zur Zahl der ändernden Aufrufe, wie in /Ran79/ festgestellt worden ist. Diese Ansicht läßt sich auf verschiedene Modelle übertragen. Da die Abschätzung bereits an der unteren Grenze liegt, sind Optimierungen nur in einem begrenzten Rahmen möglich.

Nicht so einfach ist der Aufwand für die Recovery zu ermitteln. Wenn nicht besondere Vorkehrungen getroffen sind, läßt sich keine sinnvolle Schranke für den Rücksetzaufwand angeben. Ähnlich verhält es sich mit den betroffenen Berechnungsschritten.

Trotzdem ist es falsch, Recovery-Aufwand und Verlust an Aktivitäten gleichzusetzen. Diese Behauptung kann bewiesen werden am Beispiel des sogenannten "system checkpoint", der den vollständig validierten Zustand

einer Anwendung widerspiegelt. Während der Aufwand für die Recovery proportional zur Zahl der Zustandsvariablen gehalten werden kann, ist der Verlust an Berechnung von der Laufzeit nach dem "system check-point" abhängig.

Wir wollen im weiteren von dem Verlust an Berechnungen sprechen. Wenn die Zahl der betroffenen Aktivitäten zu beschränken ist, dann kann man auch die Menge der Aktionen zur Fehlerbehandlung durch geeignete Maßnahmen (z.B. durch Aufzeichnen in einem Log) vernünftig begrenzen. Die Umkehrung der Aussage ist aber nicht immer möglich.

Das Phänomen, das die Diskussion auf dem hier umrissenen Gebiet beherrscht, ist der Domino-Effekt. Er charakterisiert den Fall, bei dem ein unakzeptabel großer Teil der Berechnung durch Recovery verloren geht. Im Extremfall kann es erforderlich werden, trotz redundanter Information alle Aktionen rückgängig zu machen.

Diese informelle Klärung des Begriffes zeigt zwar auf, welches Problem mit der Fehlerbehandlung verbunden ist, läßt aber keine formale Analyse zu.

Es ist klar, daß der Umfang der Recovery von einer Vielzahl von Parametern bestimmt wird. So sind wichtige Faktoren für die Fehlerbehandlung der Grad der Parallelität von Aktionen, die Verflechtung der Operationen untereinander, sowie der Abstand zwischen einer Veränderung und deren Validation. Diese Parameter hängen von der Gestalt des Systems ab und dürfen nicht allgemein eingeschränkt werden.

Anders verhält es sich mit der Zahl der Knoten A, die ein INVA-Graph momentan enthält. Es ist klar, daß die Recovery endet, wenn sämtliche Operationen als ungültig erkannt worden sind. Eine Abhängigkeit von der Mächtigkeit von A würde also Recovery von beliebigem Umfang erlauben. Damit ist auch schon die Größe iden-

tifiziert, von der die Zahl der betroffenen Aktionen nicht abhängen darf.

Dieses Kriterium schließt nicht aus, daß es noch andere Einschränkungen gibt, die für die Recovery in realen Systemen notwendig sind. Einige werden in den folgenden Abschnitten angesprochen. "Beschränkt" soll aber in diesem Kapitel bedeuten, daß die Größe endlich und unabhängig von der Laufzeit des Systems ist.

Unter dieser Prämisse läßt sich der Begriff "domino-frei" festlegen.

Definition 7.8:

Ein System S heißt domino-frei, wenn jede Art von Recovery ausgehend von einer beschränkten Zahl von Knoten AI aus einem beliebigen INVA-Graphen A bezüglich der Menge der rückgesetzten Aktivitäten endlich und unabhängig von der Mächtigkeit von A ist.

Für die Operationen ai , die in der Menge AI zusammengefaßt werden, muß gelten, daß es keine aj und v gibt mit $INF(ai, aj, v)$, d.h. es handelt sich um "Endknoten" von A .

Diese Definition betont den globalen Aspekt der Recovery. Da es sich bei dem INVA-Graphen um ein dynamisch wachsendes Gebilde handelt, fällt es schwer, jede Situation zu erleuchten und festzustellen, ob das System domino-frei ist. Es müssen alle möglichen Abwicklungen eines Systems, die sich in verschiedenen INVA-Graphen äußern, untersucht werden.

In der Praxis könnte das so aussehen, daß man den Verlauf einer Berechnung untersucht und dabei prüft, ob die Zahl der bei einer Recovery verlorengehenden Aktionen mit A wächst. Dieses Verfahren ist jedoch bei komplexen Systemen nicht zu realisieren.

Vor allem wenn man bedenkt, daß Recovery nicht in ge-

wöhnlichen Situationen benötigt wird, sondern eher in Ausnahmefällen angestoßen wird, ist der oben skizzierte Weg nicht gangbar.

Wünschenswert wäre, wenn man aus der Systemdefinition ableiten könnte, wie sich die Anwendung in Recovery-Fällen verhalten wird. Es liegt aber in der Natur der Fehlerbehandlung, daß die Dynamik im Vordergrund steht. Eine vernünftiger Kompromiß liegt wohl darin, lokal begrenzte Abwicklungen zu analysieren. Dieser Ansatz ist auch deshalb vorteilhaft, weil man dabei nicht auf die konkrete Realisation von Synchronisation, Abhängigkeit und Validierung eingehen muß.

Ziel der nun folgenden Überlegungen ist es deshalb, Kriterien zu finden, die es erlauben, auch bei Ausschnitten aus dem INVA-Graphen Entscheidungen über deren Recovery-Verhalten zu treffen.

Wünschenswert wäre eine "lokale" Formulierung, die äquivalent zu Definition 7.8 ist.

Ein Pfad von a_1 nach a_2 im INVA-Graphen heißt R1-sperrend, wenn durch Anwendung von R1 auf die Kanten des Pfades ausgehend von a_2 der Knoten a_1 nicht erreicht werden kann.

Der Pfad heißt R1-durchlässig, wenn durch wiederholte R1-Recovery von a_2 aus a_1 rückgesetzt wird.

Satz 7.9:

Ein System ist dann domino-frei, wenn gilt:

- (L1) Es gibt nur eine beschränkte Zahl von Endknoten, von denen die Recovery ausgeht(AI).
- (L2) Aus- und eingehende INF-Kanten einer Aktion a sind beschränkt.
- (L3) Jeder INF-Pfad, der R1 durchlässig ist, ist auch beschränkt.
- (L4) Bei einer Verzweigung von Punkt av aus gilt:

solange ein Pfad von a_v aus noch R_1 -durchlässig ist, enthalten andere Pfade nur jeweils beschränkt viele Aktionen.

Beweis:

Der Beweis wird konstruktiv über den Verlauf der Recovery geführt.

1. Die Recovery beginnt mit einer beschränkten Zahl von Knoten A_1 (L1), somit sind hier die Bedingungen von 7.8 erfüllt.

2.1 Ausbreitung der Recovery durch R_1

Jeder Knoten hat eine beschränkte Zahl von eingehenden Kanten (L2), deshalb ist auch die Anwendung dieser Regel auf eine feste Menge von Knoten limitiert. Die R_1 -Propagation terminiert nach einer beschränkten Zahl von Schritten wegen (L3).

2.2 Aktionen, die wegen R_2 rückgesetzt werden

Bei Schritt 2.1 können Knoten a_i erreicht werden, die mehr als eine ausgehende Kante haben. Der Pfad, bei dem Regel R_1 zur Anwendung kam, war R_1 -durchlässig. Aus diesem Grund ist die Zahl der Aktionen in jedem anderen Zweig von a_i beschränkt (L4). (L2) limitiert wiederum die Zahl der möglichen Pfade, die von einem Knoten ausgehen können.

Damit ist eine Menge von hinreichenden Bedingungen gefunden, die den Domino-Effekt vermeiden.

Es ist aber noch nicht gezeigt, ob die einzelnen Prädikate L_1 bis L_4 wirklich notwendig sind, um Definition 7.8 zu erfüllen.

Dies erforderte eine Umkehrung von Satz 7.9.

Die Behauptung läßt sich dann untergliedern in folgende Abschnitte:

(NDF heißt, es gibt einen INVA-Graphen, der nicht domino-frei ist)

(L1) $\vee$ (L2) $\vee$ (L3) $\vee$ (L4) nicht erfüllt $\rightarrow$ NDF

Diese Behauptung läßt sich mit Hilfe einfacher aussagenlogischer Regeln umformen zu

(G1) (L1) nicht erfüllt $\rightarrow$ NDF und

(G2) (L2) nicht erfüllt $\rightarrow$ NDF und

(G3) (L3) nicht erfüllt $\rightarrow$ NDF und

(G4) (L4) nicht erfüllt $\rightarrow$ NDF.

Da (L1) auch in der Definition 7.8 enthalten ist, gilt folglich auch (G1).

(G3) und (G4) lassen sich so beweisen, daß man direkt aus (L3) und (L4) die INVA-Graphen für das Gegenbeispiel ableitet.

Beispiel:

Wenn es einen Pfad gibt, der nach beschränkter Zahl von Operationen R1-durchlässig ist, so ist 7.8 nicht erfüllt, wenn die Menge der initialen Knoten der Recovery eine Aktion aus diesem Zweig enthält.

Für (G2) ist diese Art von Beweisführung nicht zulässig.

Kollorar 7.10:

Es gibt Systeme, die Knoten mit beliebiger Zahl

dere Modifikationen dieses Knotens zurückgegriffen und festgestellt, daß ein Fehler vorliegt. Wenn nun dieser zeitliche Abstand nicht bestimmbar ist und sich die Aktivitäten, die durch die ersten Abfragen beeinflusst wurden, nicht blockiert haben, ist eine Beschränkung des Recovery-Aufwandes nicht möglich.

Ähnliche Verhältnisse entstehen, wenn von einem Knoten mehrere Pfade ausgehen, die parallel zueinander bearbeitet werden. In diesem Fall muß die Synchronisation sicherstellen, daß alle gleichberechtigt behandelt werden. Keiner der Pfade darf beliebig viele Aktionen ausführen, während ein anderer nicht ablaufen kann. Eine "faire" Ressourcenzuteilung kann in diesem Fall den Domino-Effekt verhindern. Eine explizite Synchronisation ist ebenfalls denkbar. Dabei muß sichergestellt sein, daß kein Pfad durch falsche Koordinierung "verhungert". Aus dieser Diskussion heraus lassen sich die notwendigen Voraussetzungen für ein limitiertes Rücksetzen etwas besser verstehen.

Die Zahl der eingehenden Kanten bei einer Aktion als begrenzt anzunehmen ist realistisch, da die Zahl der Abfragen limitiert sein sollte. Wir legen diese Eigenschaften deshalb im folgenden immer zugrunde.

Unter dieser Prämisse stellt man fest, daß nur noch bei Schritt 2.2 des Beweises von 7.9 die Eigenschaft L2 benutzt wird. Daraus folgt, daß man bei anloger Beweisführung für 7.9 L2, L3 und L4 zusammenfassen kann zu

(L2') Für jede Aktion a soll gelten:

Solange es einen Pfad von a aus gibt, der R1-durchlässig ist, gibt es nur eine beschränkte Zahl von Knoten des aktuellen INVA-Graphen, die von a aus erreichbar sind.

Bei Einhaltung dieser Bedingung wird verhindert, daß

ein Knoten noch nicht vollständig validiert ist und bei späterer Abfrage ein Fehler festgestellt wird. L2' schließt aus, daß ein Knoten, der als dominofrei gilt, durch weitere Abfragen plötzlich zu einer unlimitierten Recovery führt.

Satz 7.11:

L1 und L2' sind notwendige und hinreichende Bedingungen, die jedes System erfüllen muß, um immer domino-frei zu sein.

Beweis:

Aus dem Beweis von Satz 7.9 läßt sich das Schema ableiten, wie bewiesen werden kann, daß L1 und L2' hinreichend sind.

Daß L1 notwendig ist, wurde bereits gezeigt.

Wir wollen nun annehmen, daß L2' nicht erfüllt ist.

Wenn ein Pfad von a aus R1-durchlässig ist, so gibt es eine Möglichkeit, Aktion a durch R1-Recovery zu erreichen.

Wegen R2 sind damit alle Aktivitäten, die von a abhängen, rückgängig zu machen.

Davon ist aber nach unserer Annahme eine nicht beschränkte Zahl von Knoten betroffen und das System erfüllt auch die Bedingungen von Definition 7.8 nicht.

Ein Nachteil von L2' gegenüber L2 - L4 ist, daß die Lokalitätseigenschaften nicht ganz so stark sind.

Während bei L2 - L4 die Pfade paarweise betrachtet werden können, ist bei L2' nur eine Analyse mit allen Pfaden möglich.

Werden die oben genannten Bedingungen nicht eingehalten, so können die Auswirkungen von Fehlern nicht immer begrenzt werden. Es ist auch möglich, daß beim Entwurf des Recovery-Verhaltens Fehler eingepflanzt werden,

die selbst zu Domino-Effekten führen. Diese kritischen Fälle müssen klar identifiziert werden, um sie umgehen zu können:

1) Seiteneffekte der fehlerhaften Operationen

Wenn statt einer Variablen v_i die Größe v_j modifiziert wird, entstehen andere Abhängigkeiten als vorher erfaßt.

2) Synchronisationsmängel

Wenn sich ein Pfad nicht blockiert, falls dies als notwendig erscheint oder eine Menge von Aktionen "verhungert", obwohl eine faire Ressourcenverteilung gefordert ist, kann dies zum Domino-Effekt führen. Kritisch ist auch, wenn die gemeinsamen Variablen unkontrolliert modifiziert und abgefragt werden.

3) Falsche Validierungen

In diesen Situationen kann es nicht nur dazu kommen, daß zu viele Aktionen zurückgesetzt werden, sondern auch manche fehlerhaften nicht mehr erfaßt werden können. Dieser Fall tritt wohl bei dieser Art von Defekten häufiger auf.

4) Die Systemdefinition erfüllt L1 und L2' nicht.

In diesem Fall wird man mit dem Domino-Effekt rechnen müssen, ist auch die Wahrscheinlichkeit gering, daß er tatsächlich auftritt.

Über die unerwünschten Seiteneffekte wurde bereits in Zusammenhang mit Schutzmechanismen und bei der Definition der Abhängigkeit gesprochen. Sie können eingeschränkt werden, wenn nur der Fehler rechtzeitig identifiziert wird.

Bei Synchronisationsmängeln braucht nicht jeder Fehler zu einem unerwünschten Rücksetzen zu führen. Die lokale Definition erweist sich hier als nützlich, weil

damit auch nur die entsprechenden Koordinierungen in dem begrenzten Bereich untersucht werden müssen.

Zu den letzten beiden Problemen läßt sich bemerken, daß man der präzisen Darstellung und Überprüfung der Recovery-Eigenschaften besondere Aufmerksamkeit schenken muß.

7.5 Sprachliche Darstellungsform

Nach den vorangegangenen Überlegungen fällt es nicht schwer, Recoveryaspekte in Spezifikationssprachen einzubringen. Als Basis dafür dient das Modell, das in /Ker82/ vorgestellt worden ist.

Wir wollen die Spezifikationssprache um Konstrukte erweitern, die es erlauben, die Abhängigkeitseigenschaft, die Validierungsrelation und Recovery-Gruppen darzustellen.

Als erstes soll untersucht werden, ob es besser ist, diese Prädikate auf der Modulebene (analog zu den VTGL- und PRIO-Relationen) oder auf der Operationenebene (wie die NBL-Bedingung) zu spezifizieren.

Für die Platzierung von INF und VAL auf der Modulebene spricht, daß es sich um Beziehungen zwischen zwei Operationen handelt (also ähnliche Beziehungen wie bei VTGL und PRIO). Dafür nimmt man als Nachteil in Kauf, daß man beide Aktionen identifizieren muß.

Aus Gründen der Homogenität wird hier die Lösung auf Modulebene bevorzugt.

Ein Problem bei der Formulierung der INF- und VAL-Beziehungen ist, daß es im allgemeinen bei dieser Art von Spezifikation Schwierigkeiten aufwirft, die einzelnen Aktionen eindeutig zu identifizieren.

Bei der Festlegung von Funktionen unterscheidet man nicht zwischen einzelnen Aufrufen derselben Operation. Die Namen der Funktionen können also nicht dazu dienen, die realisierten Aktionen zu adressieren. Auch

wenn man die aktuellen Parameter mit heranzieht, sind die Aktivitäten nicht in allen Fällen zu unterscheiden. Es ist aber für die Recovery wichtig herauszufinden, welche realisierten Aktionen nun wirklich miteinander in Beziehung stehen.

Eine Lösungsmöglichkeit für die INF-Relation wäre, implizit Beziehungen zwischen der letzten ändernden Aktion und der abfragenden Operation herzustellen. Dadurch wird aber die Übersichtlichkeit stark beeinträchtigt, da nun nicht per Deklaration klargestellt ist, welche Aktionen nun wirklich miteinander in Beziehung stehen. Man müßte sich aus allen Prädikaten einer Spezifikation mühsam diese Daten extrahieren. Aus diesem Grund gibt man INF- und VAL-Relation explizit durch die Operationsnamen mit ihren Parametern an. Wenn $INF(a_1, a_2, v)$, so wird implizit angenommen, daß damit der jeweils letzte Ausführung des Typs "a1" vor die Durchführung von a2 gemeint ist. Bei $VAL(a_1, a_2)$ wird die erste Operation a2 herangezogen, die noch nicht validiert wurde. Das entspringt der intuitiven Vorstellung, daß fast immer die kurz zuvor ausgeführten Operationen den meisten Einfluß auf das momentane Geschehen haben.

Der Datentyp "endlicher Puffer" mit den Operationen "get" und "put" als Gegenbeispiel zeigt, daß man diese Anschauung leider nicht gänzlich verallgemeinern kann (siehe auch das unten angeführte Beispiel).

Aus diesem Grund werden spezielle Datenstrukturen eingeführt ("RECOBJ"). Mit diesen Objekten ist eine vollständige Beschreibung von Abhängigkeiten und Validierungen möglich.

Aus den vorangegangenen Aussagen wird deutlich, wie stark der Einfluß der Dynamik von Systemen auf die Recovery ist. Detaillierter als bei der Synchronisation ist bei der Fehlerbehandlung auf das Ablaufgeschehen in der jüngsten Vergangenheit einzugehen.

Dazu kann auch die Kennzeichnung "LAST OPERATION" (LOP) einer Menge O nützlich sein. Damit ist diejenige Aktion gemeint, die als letzte der durch O bezeichneten Operationen beendet worden ist.

Mit s bezeichnen wir den Zustand des Objekts s nach Ausführung von a_1 bei $INF(a_1, a_2, v)$ oder $VAL(a_2, a_1, v)$. Wenn wir die Variablen nicht näher spezifizieren, so sind damit alle Veränderungen einer Aktion a_1 gemeint. Mit s' oder s wird der Zustand eines Objekts zum Zeitpunkt der Ausführung von Operation a_2 bezeichnet.

Das Sprachkonstrukt "ALTERNATIVE EFFECTS" (AEF) ist zur Kennzeichnung von Recovery-Gruppen gedacht. Auf Operationenebene können damit alternative Resultate der Aktion angegeben werden. Abgearbeitet werden die AEF's nach derselben Semantik wie die Recovery-Blöcke.

Die nachfolgenden Beispiele sollen diese Festlegungen besser veranschaulichen.

Validiert wird in beiden Beispielen durch eine separate Aktion. Die Wirkungen dieser Funktion schlagen sich nur im VAL-Teil nieder, deshalb entfallen die EFFECTS. In anderen Anwendungsfällen ist es aber denkbar, daß normale Aktionen bestimmte Veränderungen validieren. Es kann dann zu Schwierigkeiten bei der Definition des SYN-Teils kommen, wenn man Validierungsoperationen von den normalen Aktivitäten trennen muß. Deshalb soll die Realisierungsart in den Beispielen nicht verallgemeinert werden.

Als RECOBJ dienen vor allem Zählvariable. Durch diese einfachen Objekte lassen sich gut Sequenzen von Operationen kennzeichnen.

Als erstes soll ein klassisches Problem behandelt werden - konkurrierende lesende und schreibende Zugriffe auf gemeinsame Variable. Die elementaren Verhältnisse beim Zugriff auf gemeinsame Daten lassen sich damit darstellen.

Es ist im allgemeinen nicht möglich ist, mit einem lesenden Aufruf gleich die Gültigkeit der Daten zu überprüfen, wird eine eigene Validierungsoperation (vld) angeboten. Mit Hilfe von vld kann ein Prozeß eine Veränderung validieren, nachdem er sich durch Vergleich der gelesenen Information mit anderen Daten von ihrer Richtigkeit überzeugt hat.

```
MODULE read-write-storage(m: integer, T:TYP)

  k: integer; i1,i2,i: 0..m; data: T;

  RECOBJ: co: array(0..m) of integer INIT co(0..m)=0;

  s: read-write-storage AR AS array(0..m) of T;

  VTGL: (a1∈READ(i1) ^ a2 READ(i1) ^ i1=i2 ∨
        a1∈READ(i1) ∨ WRITE(i1) ^ a2∈READ(i2) ∨ WRITE(i2) ^ i1≠i2)
  /*die elementare Synchronisationsbedingung: nur lesende
  Zugriffe auf gemeinsame Variable sind erlaubt*/

  INF: (a1∈WRITE(i1) ^ a2∈READ(i2) ^ i1=i2 )
  /*lesende Aufrufe werden vom letzten write beeinflusst*/

  VAL: (a1∈VLD(i1,k) ^ a2∈WRITE(i2) ^ i1=i2 ^ 'co(i2)=k)
  /*vld validiert nach dem Lesen die write-Aktion*/

  read(s,i) -> data,k;
  EFFECTS: data=s(i) ^ k=co(i)

  write(s,i,data)
  EFFECTS: s(i)=data ^ co(i)='co(i)+1

  vld(s,i,k)
  /* Validiert den k-ten Eintrag in die i-te Komponente;
  die Wirkungen werden in VAL festgehalten */
MODULEEND
```

```
MODULE lbuf (m: integer where  $m \geq 1$ , T: TYPE)
  /* Spezifikation des Moduls "beschränkter Puffer nach
  /Ker82/ */

  t: T;

  RECOBJ: getc, putc: integer INIT getc,putc=0;

  q: lbuf  AR AS sequence of T
          INIT q = 0;

  INF: ( $a1 \in \text{PUT}(q) \wedge a2 \in \text{GET}(q) \wedge \text{'putc=getc}$ )
  /* die get-Aktion ist abhängig von der put-Aktion,
  die die Nachricht in den Puffer gelegt hat*/

  VAL: ( $a1 \in \text{VLD}(q) \wedge a2 \in \text{PUT}(q) \wedge \text{'putc=getc}$ )
  /* vld validiert den i-ten Eintrag */

  put(q,t)
  NBL:  $q < m$ 
  EFFECTS:  $q = \text{'q.t} \wedge \text{putc} = \text{'putc} + 1$ 

  get(q) -> t
  NBL:  $q > 0$ 
  EFFECTS:  $q = \text{trailer('q)} \wedge t = \text{first('q)} \wedge \text{getc} = \text{'getc} + 1$ 

  vld(q)
  /* Die Wirkungen werden in VAL festgehalten*/
MODULEEND
```

Es können dazu zwei Prozesse (Producer und Consumer) definiert werden, die eine korrekte Resynchronisation nach Fehlerfällen möglich machen. Dazu werden die Methoden zur Spezifikation von Prozessen angewandt, wie sie in /Mac82/ vorgeschlagen wurden. Dabei kann man innerhalb einer Operation mehrere NBL- und EFFECTS-Teile angeben, die bei der Durchführung sequentiell abgearbeitet werden. Mit CYCLIC deutet man an, daß die folgenden NBL-Bedingungen und die EFFECTS zyklisch sich wiederholend abgearbeitet werden. Mit "EFF" soll das Schlüsselwort "EFFECTS" abgekürzt werden.

Stellt sich bei einer Überprüfung heraus, daß falsche Ergebnisse vorliegen, so kann mit RECOVER vermerkt werden, daß ausgehend von der gerade bearbeiteten Operation die Recovery angestoßen wird.

Wenn bei einer Operation mit der CYCLIC-Klausel gearbeitet wird, so beziehen sich die INF- und VAL-Prädikate auf einen Durchlauf der NBL- und EFF-Teile.

Bei den INF- und VAL-Prädikaten des folgenden Moduls treten solche zyklischen Operationen X auf. Auf den ersten Blick könnte man meinen, daß X bei dieser Darstellung sich selbst validiert bzw. von sich selbst abhängig ist. Wenn man aber genau die vorangegangenen Festlegungen liest wird man feststellen, daß es sich dabei um jeweils die letzte Aktion vor der Durchführung der aktuellen Operation handelt. Damit wird auf diese Weise ausgesagt, daß X den unmittelbar vorangehenden Schleifendurchlauf validiert.

Es werden für die Spezifikation des nachfolgenden Moduls zwei weitere abstrakte Datentypen benötigt:

- 1) produce-actions mit den Operationen product-ready (Produkt fertig), repair-ready (Reparatur beendet), product-data (Produkt), product-repair (Reparatur) und product-purge(Produkt aussondern)
- 2) consume-actions mit der Operation consume (verbrauchen); dabei wird auch die Akzeptanz überprüft

MODULE producer-consumer(m,info)

pc AR AS record lbuf(m,info)
 produce-actions(info)
 consume-actions(info)

INF: $a1 \in \text{PRODUCE} \wedge a2 \in \text{PRODUCE} \wedge v \in \text{PC.PRODUCE-ACTIONS}$

VAL: $a1 \in \text{PRODUCE} \wedge a2 \in \text{PRODUCE} \wedge v \in \text{PC.PRODUCE-ACTIONS}$

/* diese Klauseln können entfallen, wenn sicher-
gestellt ist, daß aufeinanderfolgenden Produk-
tionsvorgänge unabhängig voneinander sind. v
gibt die Variable an, die validiert wird bzw.
die Abhängigkeit hervorruft. */

produce

CYCLIC:

NBL: pc.product-ready
EFF: pc.put(pc.product-data)
AEF: { pc.product-repair
 NBL: pc.repair-ready
 EFF: pc.put(pc.repair-data) }
AEF: pc.product-purge

consume

CYCLIC:

EFF: if pc.consume(pc.get)
 then pc.vld else RECOVER

MODULEEND

Darstellung des Systems als INVA-Graph:



Dieses System ist domino-frei.

Nach m Schritten muß P warten, bis C eine Nachricht entfernt. Danach kann P wieder einen put-Aufruf durchführen. Wenn nun die von C konsumierte Information fehlerhaft war, so wirkt sich das also im schlimmsten Fall auf die letzten $m+1$ Zyklen von P aus.

8. Recovery-Modelle für Prozeßsysteme

In diesem Abschnitt werden Vergleiche verschiedener Recovery-Konzepte angestellt und in das vorgestellte Schema eingeordnet.

In Kapitel 6 wurde bereits eine Methode zur Behandlung von Fehlern vorgestellt, die konkurrierende Zugriffe auf eine zentrale Datenbasis berücksichtigt. Im Sinne des INVA-Modells aus Abschnitt 7 erweist sich das Transaktionskonzept als domino-frei, weil nur R3-Recovery notwendig ist. Dies wird durch die Atomaritätseigenschaft dieses Schemas erzielt, die keine Interferenz erlaubt und dadurch, daß die innerhalb einer Transaktion erzeugten Informationen erst bei erfolgreichem Abschluß von anderen benutzt werden dürfen. Betrachtet man das Transaktionsmodell etwas genauer, so stellt man fest, daß der dabei verwendete Synchronisationsmechanismus - das Sperrprotokoll - sehr einfach gestaltet ist. Versuche, ihn auf typische Koordinierungsprobleme wie das Erzeuger-Verbraucher-Problem anzuwenden, zeigen jedoch deutlich seine Grenzen auf.

Ein erfolgversprechender Ansatz für Anwendungen mit aufwendiger Synchronisierung muß von einem allgemeineren Konzept ausgehen, vor allem wenn Fehler sich über mehrere Prozesse ausdehnen können. Bei den verteilten Systemen ist die Strategie des Transaktionskonzeptes oft nur mit großen Leistungseinbußen zu verwirklichen. Das liegt darin begründet, daß Anwendungen solange blockiert werden, bis alle Teilaktivitäten erfolgreich abgeschlossen sind.

Verallgemeinert man aber das Synchronisationsprinzip, so nimmt man in Kauf, daß der Domino-Effekt auftre-

ten kann. Mit Hilfe von konstruktiven Methoden oder Einschränkungen bei Synchronisation, Recovery und im Zusammenspiel beider Mechanismen wird versucht, dieses Problem zu lösen.

Die in diesem Kapitel dargestellten weiteren Lösungsansätze erhärten diese pauschale Aussage und zeigen an, welche Realisierungsmöglichkeiten dafür diskutiert worden sind. Zur Erläuterung wird auf das im vorangegangenen Abschnitt beschriebene Modell zurückgegriffen.

8.1 Recovery in Nachrichtensystemen

/Rus80/ bietet einen Vorschlag zur Fehlerbehandlung in Systemen an, die zur Synchronisation alleine Nachrichten zwischen den Prozessen benutzen. Der Sender hinterlegt Informationen in einer potentiell unendlichen Warteschlange, die verschiedene Prozesse in einer atomaren Operation lesen und entfernen (RECEIVE). Die Daten werden in der Reihenfolge angeboten, in der sie durch die SEND-Aktionen eingefügt wurden.

Als zusätzliche Sprachkonstrukte stehen MARK und PURGE zur Verfügung. Mit dem ersten Kommando wird der Zustand eines Prozesses gerettet, mit dem anderen Befehl der Sicherungspunkt wieder aufgelöst. Mit diesen relativ einfachen Elementen lassen sich nun einige Recovery-Probleme anschaulich aufzeigen und Lösungen für den oben angesprochenen Domino-Effekt finden.

Da im ersten Ansatz noch keinerlei Optimierungen des Aufwandes für die Speicherung der Recovery-Daten betrachtet werden sollen, brauchen wir uns um die PURGE-Operation vorerst nicht zu kümmern.

Weil Nachrichten die einzigen gemeinsamen Objekte im System sind, genügt es, zusätzlich zum Zustand der einzelnen Prozesse noch die Bewegungen in den Nachrichtenpuffern aufzuzeichnen. In "Caches", die jedem

Prozeß zugeordnet werden, sind neben den Sicherungspunkten die Interaktionen mit den Kommunikationswarteschlangen aufzunehmen. Jedem Nachrichtenpuffer ist eine Chronik ("history") zugeordnet, wor welche Informationen hinterlegt oder entnommen hat.

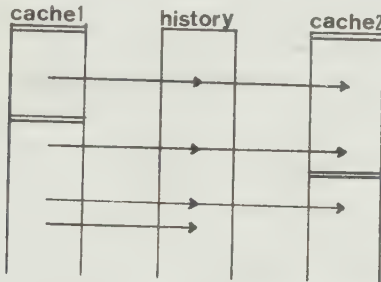


Abbildung 8.1:
Darstellung des Nachrichtenaustausches

Tritt ein Fehler auf, so wird der verursachende Prozeß auf den nächsten Sicherungspunkt zurückversetzt. Alle bis dahin gelesenen Nachrichten werden in den Puffer zurückgelegt, gesendete Informationen sind aus den Warteschlangen zu entfernen. Dadurch kann es erforderlich sein, daß andere Prozesse, die diese Nachricht bereits verwertet haben, ebenfalls von der Recovery betroffen sind (M-Propagation). Dann muß man diesen Prozeß bis zu der letzten Mark-Anweisung vor dem Empfang zurückversetzen. Auch wenn gelesene Nachrichten zurückgegeben werden, gibt es Fehlerausbreitung (R-Propagation), wenn andere Prozesse bereits weitere Informationen aus dem Nachrichtenpuffer entfernt haben. Diese Art von Fehlerausbreitung ist nur dann zu betrachten, wenn die Reihenfolge, in der die Nachrichten verarbeitet werden, Bedeutung erlangt. Beispiele für M- und R-Propagation sind Abbildung 8.2 zu ent-

nehmen.

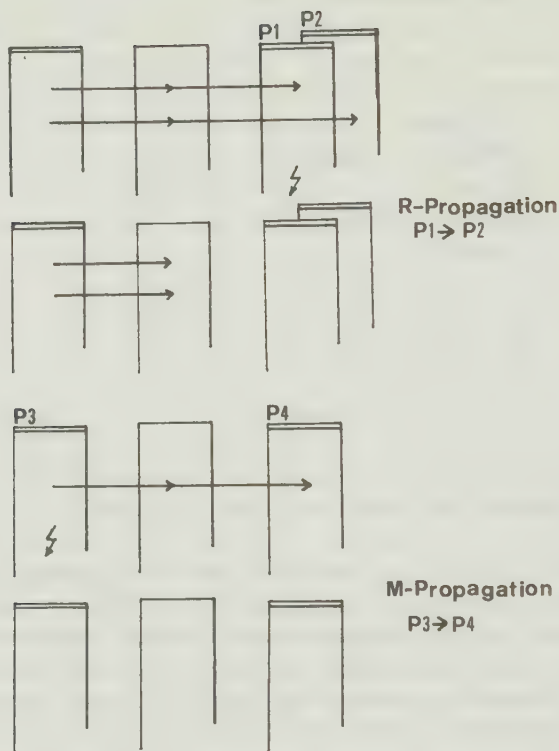


Abbildung 8.2:
R- und M-Propagation

Es fällt auf, daß dieses Recovery-Modell nur Fehler in der gerade aktuellen Bearbeitung zuläßt. Eine Ausbreitung wegen empfangener Nachrichten mit fehlerhaftem Inhalt auf den Sender ist nicht möglich. Diese Eigenschaft, die bei der Fehlerbehandlung in realen Systemen einen breiten Raum einnimmt (Gewährleistung des Produzenten), wird also in diesem Modell nicht

behandelt.

Von diesem Konzept ausgehend kann man Bedingungen ableiten, deren Erfüllen den Aufwand für das Rücksetzen beschränken.

Definition 8.1:

Wenn es pro Nachrichtenpuffer nur je einen Empfänger gibt, so heißt das System R-normal.

Lemma 8.2:

Bei R-normalen Systemen tritt die R-Propagation nicht auf.

Der Beweis ist direkt aus der Definition von R-Propagation abzuleiten.

Abbildung 8.3 zeigt, wie man R-normale Systeme aufbauen kann.

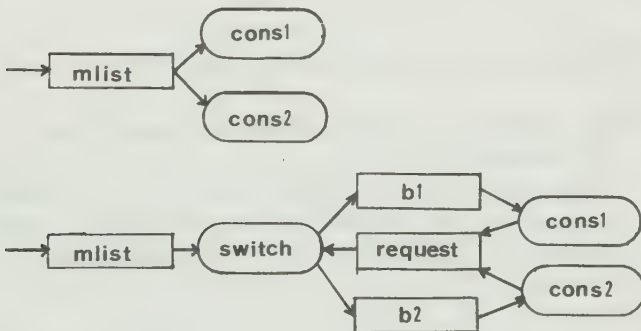


Abbildung 8.3:

Schema zum Umwandeln in eine R-normale Form

Als Domino-frei gilt auch hier das System dann, wenn

der Aufwand für die Recovery unabhängig von der Betriebsdauer ist. Dabei soll gelten, daß die Zahl der System-Aktivitäten, die parallel zu einer Menge von aufeinanderfolgenden SEND-Aufrufen eines Prozesses abgewickelt werden kann, beschränkt ist. Diese Bedingung ist in der Realität meist einzuhalten, da die Nachrichtenübergabe im allgemeinen nicht mit unbestimmbaren Verzögerungen verbunden ist. Die Zahl der Aktivitäten zwischen zwei MARK's sei beschränkt.

"*" beschreibt im folgenden eine beliebige Wiederholung des vorangegangenen, elementaren Ausdrucks.

Satz 8.3:

Wenn ein System R-normal ist, und sich die Reihenfolge der Aktionen MARK, SEND und RECEIVE eines jeden Prozesses durch den regulären Ausdruck

$(\text{MARK}; \text{RECEIVE}^* ; \text{SEND}^*)^*$

beschreiben lassen, dann ist das resultierende System domino-frei.

Beweis:

Wir können zwei Fälle unterscheiden:

- a) Der Fehler ist vor dem ersten SEND nach einem MARK aufgetreten.

Dann sind nur die RECEIVE-Aufrufe rückgängig zu machen. Nach Lemma 8.2 gibt es aber in diesem Fall keine Ausbreitung der Recovery auf andere Prozesse.

- b) Es sind SEND-Aufrufe von der Recovery betroffen.

In diesem Fall kann sich die Fehlerbehandlung auch auf andere Prozesse ausdehnen, die bereits diese Nachrichten konsumiert haben. Die entsprechenden RECEIVE-Aufrufe wurden aber erst nach

den zugehörigen SEND abgegeben. Die Folgeaktivitäten nach diesem Empfangsaufrufen sind nach unseren Annahmen beschränkt. Die vorangehenden Operationen bis zum letzten MARK sind ebenfalls nach den Voraussetzungen limitiert und führen nach Lemma 8.2 zu keiner weiteren Recovery. Auf Prozesse, die wiederum nun zurückgenommene Nachrichten konsumiert haben, lassen sich die gleichen Überlegungen anwenden, da auch sie ihre Information nach dem fehlerhaften auslösenden SEND erhalten haben.

Daß Satz 8.3 nicht gilt, wenn das System nicht R - normal ist, läßt sich gut am Beispiel in Abbildung 8.4 veranschaulichen.

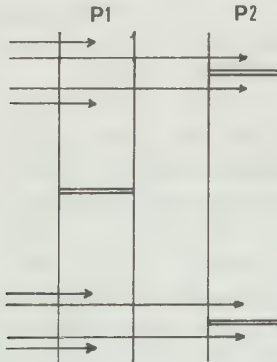


Abbildung 8.4:
Domino-Effekt durch R-Propagation

Das hier vorgestellte Modell erlaubt nur Recovery im Sinne von R2 und R3 des INVA-Konzeptes. Welche Recovery-Punkte benötigt werden, läßt sich einfach abschätzen. Setzt die Fehlerbehandlung einen Prozeß auf die MARK-Anweisung m1 zurück, so wird bei ei-

nem anderen Prozeß im schlechtesten Fall der letzte vor m1 liegende Recovery-Punkt benötigt.

Die Einschränkungen, die dieses Modell auferlegt, sind zwar für manche Anwendungen praktikabel, können aber auch in dem einen oder anderen Fall zu stark sein oder gar unerfüllbar werden.

So ist die Aussage falsch, daß ein Prozeß eine Menge von SEND-Aufrufen in einer klar definierbaren Zeit abwickeln kann, wenn

- der Prozeß durch höherpriorie Anforderungen beliebig lange verzögert werden kann (d.h. unterschiedliche Prioritäten darf es in solchen Systemen nicht geben oder sie dürfen nicht wirksam werden)
- der SEND-Aufruf mit Betriebsmittelanforderungen verbunden ist (z.B. Speicherplatzbelegung), die zu einer beliebig lange andauernden Blockierung des anstoßenden Prozesses führen können.

Die in Satz 8.3 formulierten Einschränkungen sind für manche Fälle zu stark und führen deshalb zu einer Belastung des Systems durch manche überflüssige MARK-Anweisung.

Der wohl schwerwiegendste Nachteil dieses Modells ist, daß eine Ausbreitung der Recovery von einem Empfänger auf den Sender nicht möglich ist. Wenn also nach einem RECEIVE erkannt wurde, daß diese Nachricht Fehler aufweist, so ist es nicht möglich, den Verursacher des Defektes zur Korrektur zu zwingen. Es ist aber in vielen Fällen für den Produzenten einer Nachricht unmöglich, diese Information vollständig zu prüfen, da die notwendigen Daten nur dem Empfänger zur Verfügung stehen oder weitere Nachrichten dazu gebraucht werden.

8.2 Ereignisgraphen

Wenn auch das Nachrichtenmodell sehr anschaulich und praxisnah war, eine formale Behandlung ist jedoch etwas schwierig. Deshalb wird hier noch ein zweites Modell vorgestellt, das zwar keine so klaren Beziehungen zur realen Welt hat, dafür leichter formal zu behandeln ist.

Ereignisgraphen (siehe auch Kapitel 1.3) stellen den Verlauf einer Berechnung dar. Sie wachsen mit der Zahl der Verarbeitungsschritte dynamisch. Die Knoten repräsentieren Ereignisse (Aktivitäten), die im Ablauf auftreten. Die gerichteten Kanten geben Abhängigkeiten an. Existiert eine Verbindung von a nach b, so folgte auf a das Ereignis b. Die Kanten, auch Bedingungen genannt, ordnen die Knoten so, daß keine Zyklen auftreten.

Es gibt Pfeile ohne Endpunkt, die aktiven Bedingungen. Eine einfache Interpretation dieses Modells wurde bereits in Kapitel 1.3 vorgestellt. Nach den vorausgegangenen Aussagen um INVA-Modell können wir eine Verbindung auch als INF-Kante auffassen.

Für die Recovery gibt es eine besondere Art von Kanten, die wiederherstellbaren oder restaurierbaren Bedingungen. Mit ihrer Hilfe wird der Aufwand für die Fehlerbehandlung beschränkt. Anders als beim INVA-Modell werden diese Kanten bereits bei ihrer Entstehung als restaurierbar gekennzeichnet. Eine Ausbreitung der Recovery auf das Ereignis, von dem aus diese Verbindung ausgeht, ist auf diese Weise a priori ausgeschlossen.

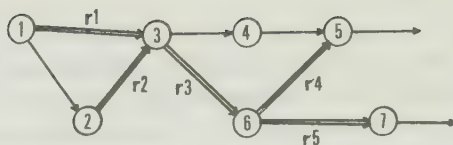


Abbildung 8.5:
Ereignisgraphen mit restaurierbaren Bedingungen

Tritt ein Defekt bei einer Teilmenge der aktiven Bedingungen auf, so muß der Graph nach den Regeln B0 bis B3 rekonfiguriert werden.

Zur Menge *INVAL* der Knoten und Kanten, die aus dem Ereignisgraphen zu entfernen sind, gehören:

- (B0) alle defekten aktiven Bedingungen.
- (B1) wenn b aus *INVAL* und e ist Ausgangspunkt dieser Kante, dann gehört auch e zu *INVAL*.
- (B2) wenn e aus *INVAL* und $(e, e') = b$, dann gehören auch b und e' zu *INVAL*.
- (B3) wenn e' aus *INVAL*, $(e, e') = b$ und b ist nicht restaurierbar, dann gehören auch b und e' zu *INVAL*.

Die Komplexität des Ereignisgraphen ist reduzierbar durch Zusammenfassen unwichtiger Zustände. Dadurch läßt sich schon im laufenden Betrieb der Aufwand für nachfolgende Recovery-Maßnahmen einschränken.

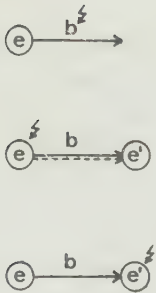


Abbildung 8.6:
Recovery-Regeln

Folgende Regeln erlauben eine Umformung des Ereignisgraphen:

(C1') Fasse Knoten zusammen, die durch keine wiederherstellbaren Kanten miteinander verbunden sind. Führt dabei eine nichtrestaurierbare Bedingung zu einem dritten Ereignis, so ist die resultierende Verbindung zu dem neu entstandenen Knoten ebenfalls nicht restaurierbar.

(C2') Ziehe alle Knoten eines Zyklusses zu einem einzigen Ereignis zusammen.

Der resultierende Ereignisgraph hat dieselben Recovery-Eigenschaften wie der ursprüngliche. Wenn also die Fehlerbehandlung eine restaurierbare Kante reaktiviert, so wird auch im umgeformten Graphen die entsprechende Bedingung wieder gültig.

Sind alle Möglichkeiten der Vereinfachung durch C1' und C2' ausgeschöpft, so entsteht ein Basisgraph, der Auskunft über die Wirksamkeit einzelner Maßnahmen zur Begrenzung von Fehlern zu einem bestimmten Zeitpunkt gibt.

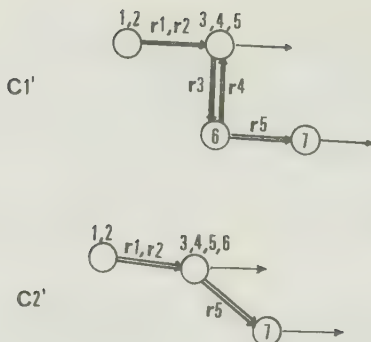


Abbildung 8.6:

Anwendung von C1' und C2' auf den Ereignisgraphen aus Abbildung 8.5

So kann man erkennen, daß es in diesem Augenblick eine Reihe von restaurierbaren Kanten gibt, die nicht in dem resultierenden Graphen erscheinen, sie sind also zur Zeit überflüssig (r_3, r_4). Die anderen noch existierenden Bedingungen bezeichnet man entweder als "sicher", wenn das Rücksetzen spätestens bei dieser Bedingung endet, egal welche der aktiven Kanten den Recovery-Vorgang auslöste (r_1, r_2), oder als "unsicher"; sie werden nur bei Versagen bestimmter aktiver Übergänge wieder gültig (r_5). Wenn (a,b) und (b,c) sichere Bedingungen sind, so ist (a,b) ebenfalls überflüssig, da die Recovery spätestens bei (b,c) endet.

Nicht festgelegt wurde in /Bes81/, was Knoten und Kanten nun wirklich bedeuteten. Wie müssen zwei Ereignisse aufgebaut sein, daß eine Kante zwischen ihnen erforderlich wird? Eine Antwort könnte - unter gewissen Einschränkungen - das in Kapitel 7 vorgestellte Modell gehen.

Will man mit Hilfe von B1, B2 und B3 Recovery durch-

führen, so ist keine zentrale Bearbeitung des Ereignisgraphen notwendig. Dieses Modell unterstützt Recovery in verteilten Systemen. Die Interaktionen zwischen den Komponenten werden dabei nicht nur zum Datenaustausch verwendet, sondern auch zur Fehlerbehandlung herangezogen. Dabei wird zur Steuerung des Ablaufs das "Chase"-Protokoll von /Mer77/ in leicht modifizierter Form verwendet:

(Ch1) wenn eine Kante für ungültig erklärt wird, so reaktiviert sich deren Anfangsknoten.

(Ch2) wenn ein Knoten reaktiviert wird und alle eingehenden Kanten restaurierbar sind, so setze man diese Bedingungen aktiv, anderenfalls erkläre alle ein- und ausgehenden Kanten für ungültig.

(Ch3) wenn ein Knoten reaktiviert wird, so werden alle von ihm ausgehenden Kanten für ungültig erklärt.

Damit die Recovery nach endlicher Zeit abbricht, ist neben genügender restaurierbarer Kanten auch noch erforderlich, daß sich die Fehlerbehandlungsaktivitäten schneller ausbreiten als die normalen Operationen. Ist das nicht möglich, so können immer neue Ereignisse und Bedingungen entstehen, die nachfolgend wieder aus dem System entfernt werden müssen.

Hier fällt auf, daß eine Ausbreitung der Fehler auch über eine Art von "R1-Regel" im Sinne von Kapitel 7 möglich ist. Die Syntax des "Chase"-Protokolls kommt dem sehr nahe, doch ist die Semantik zu schwach für eine Ausbreitung auf fehlerhafte Bedingungen. Während man durch die VAL-Relation im späteren Verlauf von Aktionen prüfen kann, ob eine vorher abgelaufene Aktivität korrekte Ergebnis lieferte, wird eine restaur-

rierbare Bedingung schon bei ihrer Entstehung als solche manifestiert. Obwohl also die Syntax dieser beiden Modelle sehr ähnlich ist, unterscheiden sich die Konzepte in ihrer Auslegung. Trotzdem lassen sich manche Eigenschaften und Erkenntnisse von dem einen System auf das andere übertragen. Ein Beispiel dafür ist die Definition des "Chase"-Protokolls für Recovery in verteilten Systemen.

8.3 Monitor-Recovery

Mehrfach untersucht wurde im Zusammenhang mit Recovery auch das Monitor-Konzept (/Hoa74/) als zugrundeliegender Synchronisationsmechanismus (/Kim78/, /Kim82/ und /Kan81/).

Vor allem interessant erscheint die Arbeit von Kant, da er auch auf das Problem eingeht; wie Recovery auf Produzenten von fehlerhaften Informationen ausgebreitet werden kann.

Die einzigen gemeinsamen Objekte sind die im Monitor definierten Variablen. Nur über sie kann ein Prozeß fehlerhafte Daten einem anderen zuspielen. Diese Größen werden unterteilt in:

1) die lokal verifizierbaren Variablen (LV).

Diese Elemente kann der Monitor selbst auf ihre Gültigkeit hin untersuchen. Über sie ist also keine Fehlerausbreitung über mehrere Prozesse hinweg möglich. Typische Vertreter dieser Klasse sind vor allem Zeiger, Zähler und andere Verwaltungsinformationen.

2) die nicht lokal verifizierbaren Variablen (NLV).

Das sind die Variablen, die nicht innerhalb der Monitorprozedur validiert werden können. Sie müssen außerhalb des Monitors oder bei folgenden Auf-

rufen überprüft werden. In Situationen, wo ein Monitor nur als "Informationsträger" dient oder in denen nacheinander von verschiedenen Prozessen Daten angeliefert werden, die dann zu komplexen Bedingungen zu verknüpfen sind, wird man auf die NLV-Variablen zurückgreifen müssen.

Auch die Struktur der Monitore soll eingeschränkt werden, um nicht die ganze Vielfalt an Realisierungsmöglichkeiten betrachten zu müssen. Trotz der Restriktionen lassen sich wohl viele Probleme noch befriedigend lösen. Ein Monitor-Prozedurrumpf soll also wie folgt aufgebaut sein:

```
begin  
  
    statement1 (LV-Variable);  
  
    if condition (LV-Variable) then wait;  
  
    statement2 (LV-Variable, NLV-Variable);  
  
    signal;  
  
end;
```

Um die Abhängigkeiten der verschiedenen Monitorbenutzer voneinander darstellen zu können, wird das READ-WRITE-Modell verwendet. Eine Abhängigkeit zwischen zwei Prozessen P1 und P2 besteht dann, wenn P1 eine NLV-Variable verändert und P2 diese über einen Monitoraufruf liest. Zwischen der WRITE- und der READ-Aktion darf es keine weiteren schreibenden Zugriffe auf diese Variable geben. Wir gehen in weiteren Überlegungen davon aus, daß jeder Monitor genau eine NLV-Variable hat.

Damit nun ein System von Prozessen und Monitoren vernünftige Recovery nach diesem Konzept erlaubt, sind folgende Bedingungen einzuhalten:

(C1) Fehlerhafte Information kann nur durch eine beschränkte Zahl von Prozessen nicht erkannt werden. Die Fehlerausbreitung endet also spätestens nach einer festen Zahl von Kommunikationsschritten ("worst-case-error-propagation-limit" - WCEPL). In unserem Fall sei WCEPL=1, eine größere Zahl ist allerdings auch denkbar.

(C2) Lese-und-Schreib-Aufrufe (Updates) von NLV-Variablen sind nicht erlaubt. Eine Monitor-Prozedur kann also diese Größen bloß lesen (READ) oder verändern (WRITE), nicht aber beides. Das Problem bei Updates ist, daß außerhalb des Monitors sonst nicht mehr festgestellt werden kann, woher die fehlerhafte Information stammt - vom letzten oder vorletzten Aufruf.

(C3) Die Recovery-Blöcke, die eine Menge von Aktionen umfassen, haben bezüglich des Zugriffs auf alle NLV-Variablen die Form

WRITE* ; READ*

Das zwingt die Prozesse dazu, gelesene Information zuerst zu prüfen, bevor sie mit diesen Daten verändernd auf weitere NLV-Variablen zugreifen.

(C4) Recovery-Blöcke enthalten eine beschränkte Zahl von Monitor-Aufrufen. Dadurch wird verhindert, daß ein Block alleine schon den Domino-Effekt erzeugt.

- (C5) Jeder Prozeß hat eine endliche Ausführungsgeschwindigkeit. Nach einer definierten maximalen Zeit ist also jede Operationenfolge abgeschlossen. Da Aktionen durch Monitore blockiert werden können, sind Maßnahmen zur Deadlockauflösung erforderlich. Dies ist notwendig, damit nach endlicher Zeit wieder ein Akzeptanz-Test ausgeführt wird.
- (C6) Die Gültigkeit einer Bedingung läßt sich überprüfen durch Vergleich einer festen Anzahl von Informationsstücken.
- (C7) Es existieren System-definierte Sicherungspunkte (SDCP) unmittelbar vor einem Monitoraufruf, wenn
- a) kein vom Benutzer definierter Sicherungspunkt (UDCP = Beginn eines Recovery-Blocks) vor dieser Operation existiert.
 - b) dies der erste Aufruf nach einem "Monitor-Call" eines anderen Prozesses ist, der die NLV-Variable ändert.

Ein SDCP dient nur zur Unterstützung einer korrekten Resynchronisation bei der Recovery, verhindert allerdings nicht, daß Fehlerbehandlung auf einen WRITE-Aufruf ausgedehnt werden kann, falls dies notwendig wird.

C7 in dieser Version liefert nicht eine minimale Anzahl von SDCP, sondern sorgt für einen möglichst geringen Recovery-Aufwand.

Grundlage für den Beweis von Theorem 8.7 sind folgende Lemmatas und Definitionen:

Definition 8.4:

Man schreibt $P_i \rightarrow M \rightarrow Q_j$, wenn P_i und Q_j Aufrufe an den Monitor M aus Prozeß P bzw. Q sind und wenn das Rücksetzen von P_i auch eine Ausbreitung der Recovery auf Q_j bewirkt. Wenn aber P_i und Q_j verschiedene Monitore ansprechen, so drückt man denselben Sachverhalt durch $P_i \rightarrow Q_j$ aus.

Lemma 8.5:

$P_i \rightarrow M \rightarrow Q_j$ gilt genau dann wenn

(1) $t(P_i) < t(Q_j)$ (d.h. P_i wird vor Q_j durchgeführt)
und ($P=Q$ oder P_i verändert M)

(2) $t(P_i) < t(Q_j)$, P_i ist ein lesender Monitoraufruf und Q_j ist ein schreibender Aufruf, der die in P_i verwendete Information produziert hat.

Lemma 8.6:

$P_i \rightarrow Q_j \Leftrightarrow \exists s, k, n (P_i \rightarrow M \rightarrow S_k \text{ und } S_n \rightarrow Q_j \text{ und } t(S_k) \leq t(S_n))$

Sei $K(Q)$ die Aktion Q_j in Q , die von einem fehlerhaften Aufruf P_i abhängt ($P_i \rightarrow Q_j$) und deren $t(Q_j)$ minimal ist.

Theorem 8.7:

Es gibt einen Prozeß R , der höchstens zwei Rücksetzschritte benötigt, um $K(R)$ zu erreichen. Es gibt aber keinen Monitor-Aufruf vor $t(K(R))$, auf den die Recovery ausgedehnt werden muß.

Damit läßt sich der Recovery-Aufwand in Grenzen halten, nicht aber die Zahl der Berechnungsschritte, die durch die Fehlerbehandlung verloren gehen. Das soll im folgenden Beispiel verdeutlicht werden. Dabei wird das in Kapitel 7 vorgestellte Modell angewandt. Zwi-

schen schreibenden und den zugehörigen lesenden Aufrufen gibt es gerichtete Kanten.

Obwohl Bedingungen C1 - C7 erfüllt sind, geht ein nicht beschränkbarer Teil der Berechnung verloren, wenn der Abstand zwischen dem Erzeugen der Information und dem Abholen nicht limitiert ist (Abbildung 8.8). Trotzdem ist die Zahl der Recovery-Schritte in diesem Fall begrenzt.

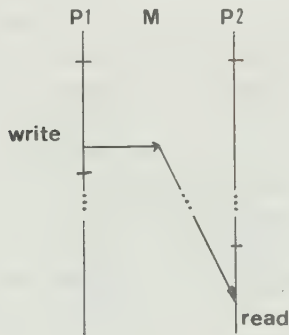


Abbildung 8.8:

Unbeschränkter Verlust, wenn Monitor M undefiniert lange Zeit nicht gelesen wird

Die Einschränkungen C1 - C7 sind in mancher Hinsicht auch zu restriktiv. So ist C3 nicht immer notwendig, um den Domino-Effekt auszuschließen. Das Beispiel in Abbildung 8.9 veranschaulicht eine Situation, wo gerade eine Verletzung dieser Vorschrift eine Recovery mit geringerem Aufwand möglich macht.

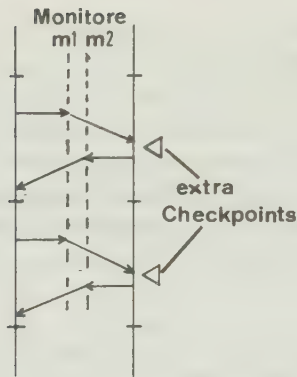


Abbildung 8.9:

Zusätzlicher Recovery-Aufwand wegen C3

Diese Darstellungen sollten zeigen, daß es keine Ideallösung für dieses Problem gibt, die sich auf alle Situationen gleich gut anwenden läßt. Nur wenn man das berücksichtigt, wird man zu vernünftigen Ergebnissen bei bestimmten Klassen von Aufgabenstellungen kommen.

9. Recovery in verteilten Systemen

Früher war man gezwungen, teure Computer unter vielen Anwendungen aufzuteilen. Da die ökonomischen Gründe für diese Organisationsform heute weitgehend weggefallen sind, erhebt sich die Frage, ob die Alternative, einzelne Lösungen auf relativ autonomen Rechnern zu installieren, nicht eher den Bedürfnissen der Benutzer entspricht.

Das Hauptargument für verteilte Systeme ist, daß auch die Organisationen, die sie benutzen wollen, zergliedert sind. Vor allem wenn eine komplexe Anwendung auch geographisch zerstreut ist, tritt ein Engpaß bei der Übertragung der Daten zwischen der Zentrale und den Außenstellen auf, wenn die Verarbeitung an einem Ort konzentriert wird (/Lis79/).

Vor allem die Forderung nach schnellem Zugriff auf lokale Daten erfordert eine verteilte Architektur. Aber auch die Möglichkeit, bei Ausfall einzelner Knoten bestimmte Funktionen weiterhin in Anspruch nehmen zu können, macht dieses Konzept attraktiv. Die Kontrolle über kleine autonome Einheiten ist leichter zu realisieren als über eine Großanlage ein perfektes Schutznetz zu legen.

Ein großer Vorteil ist auch die leichte Erweiterbarkeit des Systems, ohne daß man die gesamte Anlage stilllegen muß.

Diese positiven Eigenschaften können aber nur dann genutzt werden, wenn auch die Struktur der Software darauf abgestimmt ist. Wenn Anwendungen das nicht berücksichtigen, dann nützt auch die effizienteste Art der Kommunikation unter den autonomen Einheiten nur wenig. Wenn allerdings Synchronisation und Fehlerbehandlung zentralistisch orientiert sind, so kann ein

Benutzer wohl nur mit großen Mühen seine Programme auf verteilten Systemen effizient implementieren.

Die Betriebssystemdienste und die verwendeten Sprachkonstrukte müssen also auf die Architektur der Anlage eingehen können.

Diese Pauschalforderungen sollen anhand von Beispielen erläutert werden.

1) Eine Komponente A will Informationen an B weiterleiten, erwartet aber keine Rückmeldung. Eine Rückkopplung sollte man also vermeiden. Wenn aber A blockiert werden muß, bis B auf die Nachricht reagiert, so ist das bei langen Übertragungszeiten nicht mehr tragbar. Zur sinnvollen Fehlerbehandlung und zur Lastbegrenzung im Netz ist es aber sinnvoll, Quittierungen von Nachrichten einzuführen. Das aber sollte A nicht daran hindern, wenigstens noch eine Weile weiterzurechnen, bis Quittung q eintrifft. Besonders effizient erscheint dieses Prinzip, wenn der Zeitpunkt, ab dem auf q gewartet wird, sich variieren läßt.

2) Auch wenn A eine Nachricht an B schickt, und eine Antwort erwartet, ist ein Mechanismus nicht geeignet, der A solange blockiert, bis B die Rückmeldung empfängt. Dabei sollte A auch die Möglichkeit haben, mehrere Partner anzusprechen und erst dann auf die Antworten zu warten.

Diese Beispiele zeigen deutlich, daß die Modelle, die implizit ein Warten nach Nachrichten verursachen, für verteilte Systeme nur bedingt geeignet sind. Diese Aussage trifft auch auf das Rendezvous-Konzept bei ADA zu (/Ich79/), wo ein Partner solange warten muß, bis der andere die Rückmeldung liefert.

Gerade die Fehlerbehandlung erfordert allerdings, kommunizierende Prozesse so zu koordinieren, daß sie nicht

undefiniert viele Aktionen ausführen, bis Informationen validiert sind (siehe dazu auch Satz 7.9 und 7.11). Diese Eigenschaft darf man aber nicht so implementieren, daß die beiden Prozesse implizit warten, bis die Validierung durchgeführt worden ist. Man wird zwar wegen der Recovery die Parallelität von Prozessen nicht beliebig weit treiben können, aber es muß möglich sein, in einem vernünftigen Rahmen die Leistungsfähigkeit mehrerer Prozessoren auszuschöpfen.

Wenn ein Prozeß A warten muß, so kann das mehrere Ursachen haben:

- 1) Er will Nachrichten weitergeben, die Betriebsmittel zur Kommunikation sind aber erschöpft.
- 2) Er wartet auf Empfang einer oder mehrerer Nachrichten und kann sonst keine Aktionen ausführen.
- 3) Er muß warten, um im Fehlerfall eine sinnvolle und effiziente Recovery zu ermöglichen.

Ein Warten, nur weil der Synchronisationsmechanismus dies fordert, sollte bei verteilten Systemen ausgeschlossen werden.

Es ist jedoch möglich, daß eine konkrete Implementierung der Koordinierungsmechanismen noch weitere kurzzeitige Blockierungen einführen. Vor allem im lokalen Bereich ist eine zusätzliche Verzögerung der Prozesse beim Nachrichtenaustausch tragbar. Für die Realisierung ergeben sich aber dadurch oft Vereinfachungen, die eine kostengünstige Implementierung erst ermöglichen.

Eine Kommunikation über öffentliche Netze hinweg sollte wegen der langen, oft nur schlecht zu kalkulierenden Übertragungszeiten die Wartezeiten bei der Synchronisation der Prozesse optimieren.

9.1 Anforderungen an das Recovery-Konzept

Das in Kapitel 7 vorgestellte Modell ist unabhängig von der Struktur (zentral - verteilt) des Systems. Es geht von gemeinsamen Variablen aus, die gelesen, verändert und validiert werden. Ein unterlagerter Synchronisationsmechanismus überwacht und koordiniert den Zugriff zu diesen Variablen.

Wenn nun dieses Modell für die Recovery in verteilten Systemen eingesetzt werden soll, so ist zu untersuchen:

- 1) Sind die schreibenden und lesenden Zugriffe in verteilten Systemen gut realisierbar?

Diese Operationen sind ohne weiteres auch auf verteilte Systeme zu übertragen. Zwei Aspekte können aber zu einer modifizierten Definition zwingen:

- Die Reihenfolge des Schreibens und Lesens bestimmt die Abhängigkeit der Aktionen untereinander. Diese Sequenzen sind aber von den einzelnen Knoten aus schlecht zu ermitteln. Problematisch wird es dann, wenn eine Folge von Operationen am Ziel in einer permutierten Reihenfolge ankommt, d.h. wenn sich Aktionen überholen. Diese Situation sollte man vom Modell her ausschließen. Wie aber Aufrufe verschiedener Knoten abgearbeitet werden, obliegt dem Zielobjekt. Die Nachrichten sollten "fair" behandelt werden, damit einzelne Anforderungen nicht zu lange warten müssen.
- Ein lesender Zugriff besteht im Sinne eines Nachrichtenaustausches aus zwei Aktionen - der Anforderung und der Lieferung des Wertes. Wir wollen dies hier durch zwei Aktivitäten modellieren.

Die Abhängigkeitsrelationen sollte für verteilte Systeme erweitert werden. Wenn Aktion a_1 die Nachricht n konsumiert, die von a_2 abgegeben wurde, so kann man auch das in die Relation mit aufnehmen und es gilt $INF(a_2, a_1, n)$.

- 2) Wie stellt sich die Validierungsoperation in verteilter Umgebung dar?

Soll eine Änderung validiert werden, so ist eventuell eine Übertragung dieser Information erforderlich. Da aber der Validierende nicht weiß, wann diese Nachricht ihr Ziel erreicht, gibt es für ihn dabei drei Zustände:

- Validierungsbotschaft abgegeben, aber noch nicht validiert
- Validierung durchgeführt vom Sendeknoten
- Validierungsbestätigung erreicht den Validierenden

Wegen der Anforderungen an verteilte Systeme ist ein expliziter Aufruf notwendig, der auf Bestätigung einer Validierungsnachricht wartet. Die Simultanarbeit der einzelnen Knoten würde in vielen Fällen zu stark eingeschränkt, wenn man implizit wartet, bis die Validierungsbestätigung eingetroffen ist.

- 3) Wie werden die Prozesse effizient synchronisiert?

Zentralistisch läßt sich der Domino-Effekt durch die Blockade von Aktivitäten vermeiden, bis alle auch vom selben Ausgangspunkt ausgehenden Pfade validiert worden sind. Diese einfache Lösung mancher Probleme läßt sich aber nicht so ohne weiteres auf verteilte Systeme übertragen, denn es müßte

jeder Knoten den globalen Zustand kennen und dementsprechend seine Prozesse blockieren und wieder fortsetzen. Da sich aber diese Art der Koordination kaum aufwandsarm realisieren läßt, muß man hier Kompromisse schließen. Die beste Möglichkeit, zwei Pfade zu synchronisieren, ist, die Validierungspunkte dem jeweils anderen Knoten mitzuteilen und wenn notwendig, auf die Validierungsnachricht des Partners zu warten.

4) Wie wird die Recovery-Information gesammelt?

Bisher wurde wenig darüber gesagt, wie sich redundante Informationen gewinnen lassen. Nur beim Transaktionskonzept wurde dieses Gebiet gestreift. Für die Behandlung von Softwarefehlern (error recovery) spielt dieser Aspekt keine so zentrale Rolle. Bei verteilten Systemen, wo ganze Komplexe ausfallen und ersetzt werden müssen (crash resistance), ist es wichtig, wie man die notwendigen Recoverydaten beschafft und verwaltet.

Bei zentralistischen Systemen wird die Recovery-Information im allgemeinen in einem großen Log abgelegt. Rücksetzen einzelner Aktionen oder des Gesamtsystems läßt sich mit diesem Log ausführen. Andere Arten von Recovery gibt es nicht. Will man die Fähigkeiten von verteilten Systemen voll ausnutzen, so muß auch der Ausfall einzelner Komponenten kompensierbar sein. Diese Anforderung erzwingt eine andere Anordnung der Recovery-Daten. Sie müssen ebenfalls verteilt werden, um die Übertragungstrecken zu entlasten und eine schnelle, wirksame Fehlerbehandlung zu ermöglichen. Wird der defekte Knoten von der weiteren Verarbeitung ausgeschlossen, so ist kein Auslagern von Zustandsinformationen auf andere Komponenten notwendig. Sind seine Operationen von anderen Einheiten zu

übernehmen, so muß auch die dazu notwendige Information ausgetauscht werden. Wichtig erscheint auch die Koordination von Aktionen, Erfassen und Löschen der Recovery-Daten.

Der Austausch von Log-Informationen ist nicht unbedingt schritthaltend mit der Verarbeitung notwendig. Dazu ist der Zusammenhang zwischen Log-Aufbau und dem Ablaufgeschehen neu zu definieren. Weitere Überlegungen dazu werden im Abschnitt 9.3 behandelt.

9.2 Vereinfachtes Modell

Nachdem die grundsätzlichen Probleme der Recovery in verteilten Systemen angesprochen worden sind, wird nun eine Struktur ermittelt, die es erlaubt, dominofrei und mit minimalen Wartezeiten einfache Anwendungen zu gestalten.

Dabei beurteilt man im wesentlichen nur der Aspekt der Kommunikation zwischen den Prozessen, nicht aber komplexere Zusammenhänge zwischen den einzelnen Komponenten. Bei der INF- und VAL-Relation lassen wir die Variable als dritten Parameter weg. Der sequentielle Ablauf innerhalb eines Prozesses P_k wird vereinfacht dargestellt als eine Folge von Aktionen

$ak_0, ak_1, \dots$ mit (ak_i, ak_{i+1}) aus INF.

Weitere Kanten innerhalb eines Prozesses sind zwar denkbar, ihre Auswirkungen auf das Gesamtsystem werden aber hier nicht modelliert. Die Menge A_k enthält die Aktionen ak_i , die Prozeß P_k ausgeführt hat.

Die Kommunikation zwischen Prozeß P_k und P_l läßt sich durch eine Kante zwischen ak_i und al_j darstellen.

Bei (ak_i, al_j) aus INF mit $k \neq l$ ist ak_i der Sende- und al_j der Empfangsaufruf für eine bestimmte Nachricht.

Wir gehen von der Annahme aus, daß die Informationen

in der Reihenfolge ankommen, in der sie abgegeben worden sind.

Die Validierung einer Nachricht besteht aus zwei Schritten. Zuerst schickt man an den Absender die Validierungsbotschaft, die dieser dann in einer weiteren Nachricht bestätigt. Mit der Validierungsbotschaft wird auch mitgeteilt, daß die Nachricht eingegangen ist.

Eine Nachrichtenübertragung mit Validierung besteht also aus sechs Einzelschritten:

s(n) Eine Nachricht n wird abgegeben von Prozeß k.

e(n) Ein Prozeß P1 übernimmt diese Botschaft.

v1(n) Der Prozeß P1 sendet die Validierungsbotschaft an Pk.

v2(n) Pk empfängt die Validierungsbotschaft.

v3(n) Pk sendet eine Validierungsbesätigung an P1.

v4(n) P1 empfängt die Validierungsbestätigung.

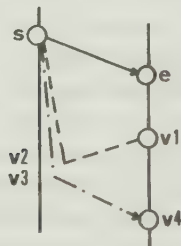


Abbildung 9.1:
Nachrichtenaustausch und Validierung

Die Aktionen des Nachrichtenaustausches finden in der Reihenfolge $s(n) \succ e(n) \succ v1(n) \succ v2(n) \succ v3(n) \succ v4(n)$ statt.

Recovery-Gruppen sollen in den ersten Überlegungen nicht betrachtet werden.

Es lassen sich Einschränkungen finden, mit denen die Recovery zu beschränken ist.

(E0) Die Zahl der kommunizierenden Prozesse ist beschränkt.

(E1) Sei $A_k = a_{k1}, \dots, a_{kh}$.

Für alle a_{ki} aus A_k gilt entweder daß

(h-i) beschränkt ist oder

es existiert ein a_{kj} mit $VAL(a_{k1}, a_{ki})$ und (j-i) ist beschränkt.

Diese Eigenschaft beschränkt die Recovery in nicht kommunizierenden Prozessen.

(E2) Wenn $a_{kj} = s(n1)$ und $a_{ki} = v3(n1)$, dann ist (i-j) beschränkt.

(E3) Wenn $a_{kj} = e(n1)$ und $a_{ki} = v4(n1)$, dann ist (i-j) beschränkt.

Diese Eigenschaften garantieren, daß Prozesse nur eine limitierte Zahl von Aktionen durchführen, bis die Validierungsbestätigung initiiert wird.

(E4) Wenn $s(n1)$ und $s(n2)$ aus A_k und ist $s(n1) \succ s(n2)$, dann gilt auch $v3(n1) \succ v3(n2)$

(E5) Wenn $a_{ki} = s(n1) \succ a_{kj}$ und $v3(n1)$ bereits durchgeführt worden ist, und es kein $s(n2)$ gibt mit $a_{ki} \succ s(n2) \succ a_{kj}$, dann existiert ein a_{kg} und ein a_{kf} mit $VAL(a_{kf}, a_{kg})$ und $i \succ g \succ j$.

(E6) Senden, Empfangen und Validieren läuft nach einem der folgenden Muster ab (alle angeführten Operationen gehören zu einem Prozeß):

$s(n2) > e(n1)$

$e(n1) > v1(n1) > v4(n1) > s(n2)$

$e(n1) > s(n2) > v2(n2) > v1(n1) > v4(n1) > v3(n2)$

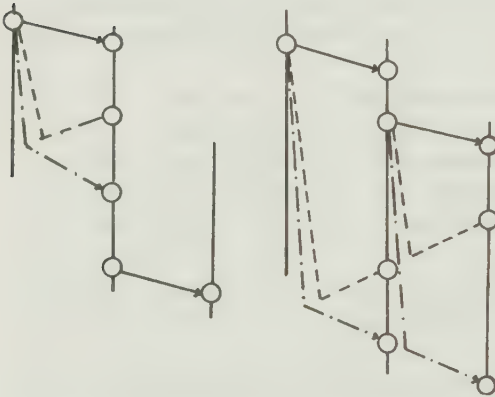


Abbildung 9.2:

Mögliche Abwicklungen von Kommunikationsaufrufen nach (E6)

Lemma 9.1:

Erfüllt ein System (E4), (E5) und (E6), so ist $s(n1)$ nicht durch Recovery mit den Regeln R1 und R2 ausgehend von einer beliebigen Menge von Endknoten erreichbar, wenn $v3(n1)$ bereits durchgeführt worden ist.

Beweis:

Es gibt zwei Möglichkeiten, $s(n1)$ zu erreichen:

a) über R1

b) über R2.

zu a) Da $v_3(n_1)$ existiert, sind auch bereits $v_1(n_1)$ und $v_2(n_1)$ durchgeführt worden. Damit ist aber bei Rücknahme von $e(n_1)$ eine Ausbreitung auf $s(n_1)$ nicht mehr erforderlich.

(E5) besagt, daß es nach $s(n_1)$ und vor der folgenden Sendeoperation eine bereits validierte Aktion existiert. Dies verhindert eine Ausbreitung auf $s(n_1)$ über R_1 durch eine der nachfolgenden Operationen.

zu b) Über einen Prozeß P_l breitet sich die Recovery auf eine Aktion a_{kj} des Prozesses P_k vor $s(n_1)$ aus. Dies muß eine Aktion zum Austausch von Nachrichten gewesen sein.

Daß es ein $s(n_2)$ aus A_k war, ist ausgeschlossen, da nach (E4) dieser Nachrichtenaustausch bereits validiert war.

War es ein Empfangsaufwurf $e(n_2)$, so muß nach (E6) $v_4(n_2)$ bereits durchgeführt worden sein.

Damit existiert auch $v_3(n_2)$ und Lemma 9.1 kann auf $s(n_2)$ angewandt werden.

Diese Rekursion terminiert, da der Graph endlich ist und $s(n_2) > s(n_1)$.

Es gibt also keine Möglichkeit, $s(n_1)$ durch die Recovery zu erreichen.

Bedeutung von Lemma 9.1:

$v_3(n_1)$ leitet den letzten Kommunikationsschritt bei einem Nachrichtenaustausch ein. Es ist im allgemeinen nicht kalkulierbar, wieviele Schritte die beiden Partner danach durchführen, bis die nächste Interaktion zwischen ihnen stattfindet. Bei einem Rücksetzen von $s(n_1)$ nach diesem Schritt ist deshalb die Zahl der

Aktionen, die von der Recovery betroffen sind, nicht beschränkbar.

Satz 9.2:

Wenn (E0) bis (E6) erfüllt sind, so ist das resultierende System domino-frei.

Beweis:

(E1) limitiert die Menge der Aktionen in Abschnitten, in denen keine Send-Aufrufe stattfinden.

1) Bis $v_3(n_1)$ ist die Zahl der von $s(n_1)$ abhängigen Aktionen beschränkt:

(E2) und (E3) beschränken die Aktionen des sendenden bzw. empfangenden Prozesses von $s(n_1)$ oder $e(n_1)$ ausgehend.

Die Kaskadierung von Sende- und Empfangsaufrufen nach (E6) ist beschränkt durch (E0) in Verbindung mit (E2) und (E3).

2) Lemma 9.1 besagt, daß alle Pfade von $s(n_1)$ aus R1-sperrend sind, wenn $v_3(n_1)$ durchgeführt wurde.

Damit ist das System domino-frei nach Satz 7.11.

Eine bedeutsame Einschränkung vermittelt 9.2, die in manchen Fällen zu unerwünschten Wartezeiten führen kann. Zwischen dem Senden der Validierungsnachricht und dem Empfang der Quittung dürfen keine Daten an andere Prozesse weitergeleitet werden. Die Regel kann man durchbrechen, wenn durch weitere Kommunikations-schritte die Prozesse domino-frei koordiniert werden können. Diese Eigenschaft zeigt deutlich, wie schwer es ist, auf allgemeiner Basis dieses Problem befriedigend zu lösen. Nur über weitere Einschränkungen oder

vermehrten Nachrichtenaustausch wird es möglich, diese Grenze weiter zu verschieben.

Dieses Schema erhöht die Deadlock-Gefahr, da zusätzliche Wartestellen eingeführt werden (siehe auch das Beispiel in Abbildung 9.3). Es ist deshalb gefährlich, diese Wartestellen vom System vergeben zu lassen.

Das einfache Modell kann man um Recovery-Gruppen erweitern. Damit erhöht sich noch einmal die Zahl der Pfade, die miteinander koordiniert werden müssen.



Abbildung 9.3:

Deadlocks durch Warten auf das Validieren

(w = Warten auf Validierungsnachricht)

Das Problem liegt darin, daß es nun nicht mehr ausreicht, die Schnittpunkte zwischen zwei Pfaden zu validieren, wie das beim Schema von Satz 9.2 vorgegeben ist. Erst wenn überhaupt keine Möglichkeit mehr besteht, die entsprechenden Recovery-Gruppen nach einem Defekt zu erreichen, können Validierungsnachrichten abgegeben werden.

Leider ist diese Art von Koordinierung in wichtigen Anwendungsfällen nicht anwendbar. Wenn zwei Prozesse P1 und P2 jeweils innerhalb einer Recovery-Gruppe eine Nachricht an den anderen abgeben und empfangen, so entsteht ein Deadlock (Abbildung 9.4).

Diese Situation tritt auch dann auf, wenn mehr als

zwei Prozesse einen Zyklus bilden. Damit sind auch die Querbezüge zwischen Recovery-Gruppe, Atomaritätseigenschaft und Deadlocks hergestellt. Erst Zyklen unter Recovery-Gruppen schaffen Probleme. Sie kennzeichnen das Auftreten von Deadlocks und legen fest, daß Aktionsfolgen nicht atomar sind.

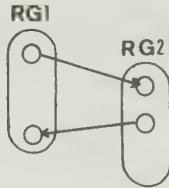


Abbildung 9.4:
Zyklen zwischen Recovery-Gruppen

Um die Probleme klarer identifizieren zu können, wollen wir den Abschluß einer Recovery-Gruppe von der Validierung separieren.

Recovery-Gruppen, die keinem Zyklus angehören, kann man mit Hilfe einiger Regeln relativ leicht dominofrei aufbauen. Die Bedingungen, die eine Recovery-Gruppe einzuhalten hat, lassen sich aus den Einschränkungen (E0) bis (E6) ableiten. Man muß nur die Recovery-Gruppe als eine Aktion betrachten. Die Validierungsbestätigung wird dabei durch eine Nachricht zum Abschluß der Recovery-Gruppe ersetzt.

Folgende Bedingungen sollten erfüllt sein, wenn ein Prozeß Pk bekanntgibt, daß seine Recovery-Gruppe g abgeschlossen ist:

- 1) die letzte Aktion von g ist validiert (dies ist eine vereinfachte Version von (E5))
- 2) sämtliche Send-Aktivitäten von g sind validiert

- 3) Recovery-Gruppen, von denen g Nachrichten empfangen hat, sind abgeschlossen
- 4) Recovery-Gruppen, die von Prozeß k vor g begonnen wurden, werden vor g abgeschlossen.

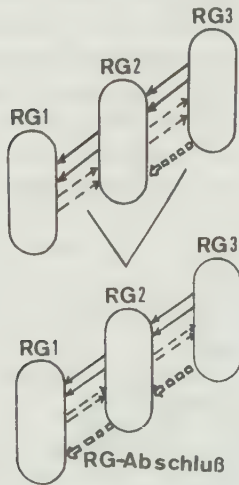


Abbildung 9.5:
Abschluß zyklischer Recovery-Gruppen

Bestehen zyklische Abhängigkeiten, so kann dieses Verfahren nicht direkt angewandt werden.

Geht man davon aus, daß bekannt ist, welche Recovery-Gruppen durch einen geschlossenen Pfad miteinander verbunden sind, so läßt sich eine Lösung für ein dominofreies Abschließen finden.

Man bestimmt zuerst eine Recovery-Gruppe oder auch einen neutralen Koordinator, die den Abschluß leiten.

Als erstes wird an alle über den Zyklus miteinander verbundenen Recovery-Gruppen eine Abschlußanforderung

geschickt. Diese melden dann zurück, ob ihre Recovery-Gruppe abgeschlossen werden könnte. In einer zweiten Runde wird dann der Abschluß effektiv durchgeführt. Dieses Verfahren entspricht dem "Two-Phase-Commitment-Protocol" (/Gra79/).

Es gibt eine Möglichkeit, die Zahl der dabei notwendigen Nachrichten zu minimieren. Man braucht nicht von einem zentralen Koordinator aus alle beteiligten Recovery-Gruppen direkt anzusprechen. Derselbe Effekt wie beim oben beschriebenen Verfahren kann erreicht werden, wenn die Abschlußanforderung und die Abschlußbestätigung innerhalb eines Zyklusses von Recovery-Gruppe zu Recovery-Gruppe weitergereicht werden und man so dieselben Wege geht wie bei allen anderen Nachrichten.



Abbildung 9.6:

Schritte zum Abschluß von Recovery-Gruppen, die zyklisch miteinander verbunden sind

(1-3: Abschlußanforderung,

4-6: Abschlußbeendigung)

Auch diese Funktionen können parallel zu aktiven Recovery-Gruppen durchgeführt werden und erfordern im Prinzip kein Warten.

9.3 Behandlung von Recovery-Daten

Wie bereits ausgeführt, gibt es auch beim Speichern von Recovery-Informationen (Logging) in verteilten Systemen Probleme. Selbst bei sequentiellen Abläufen bereiten diese Funktionen Schwierigkeiten, wenn man beliebige Ausfälle betrachtet (/Ran80/, /Gra77/). Wir wollen hier vor allem zwei Möglichkeiten näher betrachten:

- 1) Eine Komponente fällt für eine bestimmte Zeit aus und nimmt dann seine Verarbeitung wieder auf.
- 2) Aufgaben von defekten Teilen werden von mehreren anderen Knoten übernommen (graceful degradation).

Die zuerst erwähnte Möglichkeit erscheint einfacher, da auf den ersten Blick keine Koordinierung der Logs von den verschiedenen Komponenten erforderlich zu sein scheint.

Für alle veränderten gewöhnlichen Daten ist die Regel wirksam, daß Informationen erst nach einem Log-Eintrag modifiziert werden dürfen (WAL-Prinzip). Bei Nachrichten ist diese Forderung alleine nicht hinreichend. Sie durchlaufen eine Reihe von Zuständen, die von den Prozessen verschieden gesehen werden. Nachrichten identifiziert man eindeutig, damit bei der Recovery klar ist, um welche Informationen es sich handelt.

Es gibt prinzipiell zwei Möglichkeiten, wie man auf Ausfälle reagiert:

- 1) Auf alle Recovery-Gruppen, die betroffen sein könnten, wird die Fehlerbehandlung ausgedehnt. Regel R1 wird voll angewandt.
- 2) Es werden nur die direkt betroffenen Recovery-Gruppen und die davon abhängigen Aktionen zurückgesetzt.

Die erste Lösung minimisiert die Menge der Log-Einträge und erlaubt gleiche Reaktion bei Ausfällen und anderen Fehlertypen (siehe Abbildung 9.7). Die Alternative dazu minimisiert die Zahl der Rücksetzschritte bei Ausfällen. Wenn nämlich bei einem "crash" eines Empfängerknotens (Node 2) die Recovery auf den Absender ausgedehnt wird, so braucht Node 1 nur zu vermerken, daß er eine nichtvalidierte Nachricht abgegeben hat, nicht aber den Inhalt. Will man ein Rücksetzen von RG1 vermeiden, so muß der Inhalt der Nachricht ins Log von Node 1 geschrieben werden.

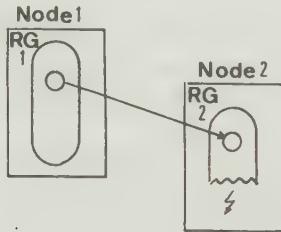


Abbildung 9.7:
Crash-Recovery

Betrachtet man die Zustände einer Nachricht genauer, so gibt es drei Aktionen, die für die Recovery von großer Bedeutung sind:

1) Das Senden der Nachricht

Von diesem Zeitpunkt ab muß man bei der Fehlerbehandlung berücksichtigen, daß auch ein potentieller Empfänger der Information betroffen sein kann, wenn ein Defekt vom Absender erkannt wird. Es ist mindestens zu vermerken, an wen die Nachricht gegangen ist bzw. gehen soll (siehe dazu auch die Bemerkungen zu Abbildung 9.7), um bei Verlust der Nachricht

beim Empfänger Recovery-Maßnahmen ergreifen zu können.

2) Empfang einer Nachricht

Nachdem die Information ihr Ziel erreicht hat, geht sie auch in die Datenstrukturen des Empfängers ein. Eine Protokollierung dieses Ereignisses bringt nur dann Vorteile, wenn man die Nachricht noch einmal benötigt und nicht auf den Absender zurückgreifen will. Da aber bis zur Übernahme der Daten beliebig viel Zeit verstreichen kann, spart man durch das Logging nur zusätzliche Übertragungen. Andererseits ist aber in den meisten Fällen die Archivierung der Nachricht früher oder später notwendig. Dies ist wohl der einzige Grund, der für eine Protokollierung der Nachricht zu diesem Zeitpunkt spricht.

3) Absenden der Validierungsnachricht

Wenn diese Aktion durchgeführt worden ist, darf kein Rückgriff mehr auf den Absender erforderlich werden, denn sonst ist bei Empfängerfehlern eine Ausbreitung auf den Sender erforderlich. Damit ist aber die geforderte Limitierung der Recovery gefährdet. Es müssen also Nachricht und Validierungsvermerk in dem Log des Empfängers hinterlegt werden. Wenn im Rahmen der Recovery auch die Validierungsoperation betroffen sein könnte, so ist diese Aktion zu wiederholen. Ist die Datenübernahme in die Recovery miteinbezogen, so kann das nach dem Validierungszeitpunkt dadurch gelöst werden, daß diese Nachricht wieder in den Puffer zurückgelegt wird.

Wenn eine Recovery-Gruppe vollständig abgeschlossen wurde, können auch die Log-Einträge über den Datenaustausch gelöscht werden.

9.4 Beispiel-System

An einem einfachen Beispiel sollen die Funktionsweisen der Recoverymaßnahmen erläutert werden. In der Form lehnt sich das hier vorgestellte System an /Lis82/ an.

Es wird eine Verteilanlage vorgestellt, bei der ein Weg von dem Eingang zu einem der Ausgänge durchgeschaltet wird, auf dem dann das zu sortierende Objekt durch das System geschleust wird (siehe Abbildung 9.9). Die dazu notwendigen Elemente unterteilen sich in drei Typen: Eingabe-(INPUT), Ausgabe-(OUTPUT) und Zwischenknoten (SWITCH).

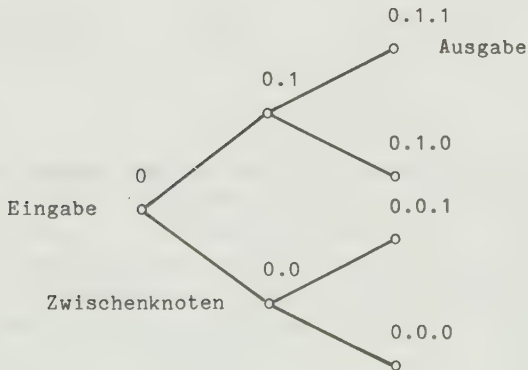


Abbildung 9.9:
Verteilanlage

process INPUT

cyclic

Lese_Zieladresse(adr);

div:=adr/K; rem:=adr mod K;

send div, 0 to 0.rem;

rcv1 on 0; /*Empfang der Validierung*/

snvb to 0.rem; /*Validierungsbestätigung abgeben*/

end-cyc;

end-p;

process OUTPUT(n) /*n gibt an um welchen Ausgang es
sich handelt*/

cyclic

recv adr,m on n

snv1 to m

rcvb on n

end-cyc;

end-p;

```
process SWITCH(n)
```

```
cyclic
```

```
    recv adr, m on n
```

```
    div:=adr/K; rem:=adr mod K;
```

```
    send div, n to n.rem;
```

```
    rcv1 on n
```

```
    snv1 to m
```

```
    rcvb on n
```

```
    snvb to n.rem /*Reihenfolge nach (E6) aus 9.2*/
```

```
end-cyc;
```

```
end-p;
```

Im Fehlerfall (Ausfall und Neustart eines Prozesses) können drei Situationen auftreten:

- 1) Als letztes wurde eine Validierungsbestätigung abgegeben.

In diesem Fall wird diese Validierungsbestätigung wiederholt. Die Datenbasis kann in dem Zustand gelassen werden, wie sie der Prozess im Log vor findet (=Zustand zum Zeitpunkt der Validierungsbestätigung). Der Ausfall wird, wie in anderen Situationen auch, dem übergeordneten Prozeß, von dem man Nachrichten empfängt, mitgeteilt.

- 2) Als letztes ist die Validierung vermerkt.

In diesem Fall muß von den übergeordneten Prozessen entschieden werden, welche Recoverymaßnahmen zu treffen sind. Dieser Knoten wird also abwarten, ob er als Reaktion auf seine Ausfallmeldung eine Rücksetzaufforderung oder eine Validierungsbestäti-

gung erhält.

3) Als letztes wurde die Nachricht weitergeleitet.

Es ist nicht immer möglich, in diesem Zustand für einen Erhalt der Nachricht zu sorgen. Aus diesem Grund wird eine Rücksetzaufforderung an die unterlagerten Knoten gesendet und die Datenbasis wird auf den Zustand zum Zeitpunkt des Nachrichtenempfangs zurückgesetzt.

Es ist zu einer korrekten Resynchronisation der Prozesse notwendig, daß die in 9.2 vorgestellte Einschränkung E6 erfüllt ist, da sonst Zwischenzustände auftreten können, die mit 1-3 nicht behandelt werden können.

10. Zusammenfassung und Schlußbemerkungen

Schwerpunkt dieser Arbeit war die Recovery, das Erreichen eines Zustands, von dem aus ein System nach Fehlern wieder sinnvoll weiterarbeiten kann. Um diese Aktivität gut zu modellieren, sind präzise Vorstellungen über das Systemverhalten im Normal- und Fehlerfall notwendig. Nur wenn man weiß, wie Defekte in Hard- und Software entstehen und wie sie erkannt werden, lassen sich Recoverymaßnahmen begründen. Es gibt Fehler, die "kalkulierbar" aber unvermeidbar sind. Für diese Klasse können eigens dafür geschaffene Routinen die Fehlerbehandlung bewerkstelligen. Anders verhält es sich mit Entwurfsfehlern, bei denen man nicht weiß, wann und wo sie auftreten. Alle Programmierfehler gehören dieser Kategorie an. Die Recovery muß dann einen konsistenten Zustand wiederherstellen, bei dem alle fehlerhaften Aktionen aus dem System entfernt sind. Zusätzlich dazu sind alternative Berechnungspfade zu realisieren, da das Durchlaufen derselben Anweisungen wieder zu Fehlern führt. Das Recovery-Block-Konzept war ein erster Schritt in diese Richtung. Es ist aber nicht anwendbar auf ein System kommunizierender Prozesse.

Wenn Synchronisation und Vorsorge für den Fehlerfall unkoordiniert ablaufen, dann läßt sich manchmal die Recovery nicht beschränken. Dieses Phänomen ist als Domino-Effekt bekannt geworden. Die Recovery ist durch zwei Relationen bestimmt. Die Abhängigkeitsrelation gibt an, wie die Einzelaktivitäten sich beeinflussen. Eine zweite Beziehung beschränkt die Recovery. Sie kann auf verschiedene Art und Weise definiert werden. In dem hier vorgeschlagenen Modell ist eine Aktion A von B abhängig, wenn A eine Variable abfragt und

B diese Größe als letztes verändert hat. Durch Validierung (=Bestätigung der Richtigkeit) wird die Recovery beschränkt. Wenn eine Veränderung validiert ist, braucht bei Defekten in den abhängigen Aktionen die Fehlerbehandlung nicht auf die modifizierende Operation ausgedehnt werden. Zur Strukturierung von Aktivitäten benutzt man Recovery-Gruppen. Einzelaktionen können damit zu komplexeren Gebilden zusammengefaßt werden. Auch für verteilte Systeme ist diese Art der Fehlerbehandlung anwendbar. Dabei ist darauf zu achten, daß die Zahl der Nachrichten, die zur Vorsorge für den Fehlerfall ausgetauscht werden müssen, minimiert wird. Kritisch sind dabei der Ausfall von Komponenten und die Vorsorge für diesen Fehlerfall, das "Logging" der Aktionen.

Viele Probleme konnten in dieser Abhandlung nicht ausführlich behandelt werden. So wurden konkrete Anwendungen des Modells nicht aufgegriffen. Bei der Lösung mancher praktischer Probleme erwiesen sich diese Überlegungen schon oft als nützlich. Vor allem die Fehler-suche in Anlagen konnte durch konsequente Anwendung dieses Modells beschleunigt werden.

Es müssen aber noch weitere Anwendungsgebiete erschlossen werden, um die Erfahrungen auf eine breitere Basis zu stellen. Dabei ist zu untersuchen, ob hinreichende Bedingungen für die Domino-Freiheit für spezielle Einsatzfälle gewonnen werden können. Mit Hilfe konstruktiver Verfahren sollten domino-freie Systeme zu entwerfen sein.

Vor allem die Beispiele bei den verteilten Systemen zeigen deutlich, das noch Vereinfachungen notwendig sind, damit eine möglichst große Zahl von Entwicklern sich diese Überlegungen zu eigen machen können.

Bei der Zustandssicherung wurden oft Zählvariable eingesetzt, um eine korrekte Resynchronisation zu ermöglichen. Es ist allerdings jetzt noch nicht möglich, diese Realisierungsart zu verallgemeinern.

11. Literatur

- /Alf77/ Alford, M.:
A Requirements Engineering Methodology for Real-Time Processing Requirements
IEEE Trans. SwE 3, 1977, pp. 60 - 69
- /And81/ Anderson, T.; Lee, P.A.:
Fault Tolerance - Principles and Practice
Prentice Hall, 1981
- /Avi75/ Avizienis, A.:
Fault-Tolerance and Fault-Intolerance:
Complementary Approaches to Reliable Computing
FTCS 1975, pp. 458 - 464
- /Ban78/ Banatre, J.P.; Shirivastava, S.K.
Reliable Resource Allocation between Unreliable Processes
IEEE Trans. SwE, 4, 1978, pp. 230 - 241
- /Bel77/ Bell, T.; Bixler, D.; Dyer, M.:
An Extendable Approach to Computer-Aided Software Requirements Engineering
IEEE Trans. SwE, 3, 1977, pp. 49 - 60
- /Bes81/ Best, E.; Randell, B.:
A Formal Model of Atomicity in Asynchronous Systems
Acta Informatica 16, 1981, pp. 93 - 124

- /Boe76/ Boehm, B.W.:
Software Engineering
IEEE Trans. Computers, 1976, pp.1226 - 1241
- /Bon80/ Bonn, G.; Patz, M.; Saenger, F.:
Grundprinzipien und Betriebserfahrung mit
Fehlererkennung und -anzeige bei fehlertole-
ranten Prozeßrechnersystemen mit funktions-
beteiligter Redundanz
Fachberichte MSR, Band 5, Springer Verlag,
1980, pp. 329 - 352
- /Che78/ Chen, L.; Avizienis, A.:
N-Version-Programming:
A Fault Tolerance Approach to Reliability
of Software Operation
FTCS 1978, pp. 3 - 11
- /Cri79/ Cristian, F.:
A Recovery Mechanism for Modular Software
IEEE Conf. SwTech., 1979, pp. 42 - 50
- /Dal80/ DalChin, M.:
Diagnostizierbarkeit und Zuverlässigkeit von
Multiprozessorsystemen
Vortrag im Rahmen des IMMD-Kolloquiums, Er-
langen, 1980
- /Den76/ Denning, P.J.:
Fault Tolerant Operation Systems
Computing Surveys, 8(4), 1976, pp. 359 - 389
- /End80/ Endres, A.:
Methoden der Programm- und Systemkonstruktion
Informatik-Spektrum, 3, 1980, pp. 156 - 171

- /Esw76/ Eswaran, K.P.; Gray, J.N.; et.al.:
The Notions of Consistency and Predicate Locks
in a Database System
CACM 19(11), 1976, pp. 624 - 634
- /Fri80/ Friedman, A.D.; Simoncini, L.:
System-Level Fault Diagnosis
COMPUTER, 3, 1983, pp. 47 - 53
- /Gla81/ Glass, R.:
Persistent Software Errors
IEEE Trans. SwE, 7, 1981, pp.162 - 168
- /Goo75/ Goodenough, J.B.:
Exception Handling:
Issues and a Proposed Notation
CACM, 18(12), 1975, pp. 683 - 696
- /Gra79/ Gray, J.N.:
Notes on Data Base Operating Systems
in: Operating Systems - An Advanced Course
Springer-Verlag, 1979, pp. 393 - 481
- /Gut76/ Guttag, J.V.; et.al.:
Abstract Data Types and the Development of
Data Structures
CACM 20(6), 1977, pp. 396 - 404
- /Här80/ Härder, T.:
Logging und Recovery in Datenbanksystemen
Vortrag im Rahmen des IMMD-Kolloquiums, Erlangen, 1980
- /Hay78/ Hayes, J.P.; Yannev, R.:
Fault Recovery in Multiprocessor Networks
FTCS 8, 1978, pp. 123 - 128

- /Hen79/ Henninger, K.:
Specifying Software Requirements for Complex
Systems:
New Techniques and their Application
Proc. Specifications of Reliable Software,
IEEE, 1979, pp. 1 - 14
- /Hoa74/ Hoare, C.A.R.:
Monitors: An Operation System Structuring
Concept
CACM 17(10), 1974, pp. 549 - 557
- /Hof76/ Hofmann, F.:
Prozeßmodelle aus der Sicht des Betriebssy-
stementwurfs
IMMD-Berichte, 9(8), 1976, pp. 233 - 246
- /Hop80/ Hopkins, A.:
Fault Tolerant System Design:
Broad Brush and Fine Print
COMPUTER, 3, 1980, pp. 39 - 45
- /Ich79/ Ichbiah, J.D.; et.al.:
Rationale for the Design of the ADA Program-
ming Language
Preliminary ADA Reference Manual
ACM SIGPLAN Notices, 6, 1979
- /Jac75/ Jackson, M.A.:
Principles of Programm Design
Academic Press, 1975
- /Jon79/ Jones, A.K.; Schwarz, P.:
Experience Using Multiprocessor Systems
CMU-Report, Pittsburg, 1979

- /Kan81/ Kant, K.:
Error Recovery in Concurrent Processes
University of Texas at Dallas, 1981
- /Ker77/ Keramidis, S.:
Ein allgemeines formales Modell für Betriebs-
systeme und dessen Implementierung
Dissertation, IMMD-Berichte 10(13), 1977
- /Ker79/ Keramidis, S.; Mackert, L.:
Specification and Implementation of Parallel
Activities on Abstract Objects
Proc. Int. Conf. SWE, 1979, pp. 203 - 211
- /Ker82/ Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
IMMD-Berichte 15(4), 1982
- /Keu82/ Keutgen, H.:
Qualitätssicherung in Software Projekten
Online 1, 1982, pp. 64 - 68
- /Kim78/ Kim K.H.:
Methodologies and Tools for Developing Ro-
bust FTSC Software
U.S. Airforce, 1978
- /Kim82/ Kim, K.H.:
Approaches to Mechanisation of the Conver-
sation Scheme Based on Monitors
IEEE Trans. SWE 8, 1982, pp. 189 - 197

- /Kop79/ Kopetz, H.:
Software Reliability
Hanser-Verlag, 1979
- /Lin78/ Linden, T.A.:
Operating System Structures to Support Security and Reliable Software
Computing Surveys, 8(4), 1978
- /Lis75/ Liskov, B.; Zilles, S.N.:
Specification Techniques for Data Abstraction
IEEE Trans. SwE 1, 1975, pp. 7 - 19
- /Lis79/ Liskov, B.:
Primitives for Distributed Computing
ACM Proc. of the 7th Symposium on Operating System Principles, 1979, pp. 33 - 42
- /Lis82/ Liskov, B.:
On Linguistic Support for Distributed Programs
IEEE Trans. SwE, 8, 1982, pp. 203 - 210
- /Lom77/ Lomet, D.B.:
Process Structuring, Synchronisation and Recovery Using Atomic Actions
SIGLAN-Notices, Vol.12(3), 1977, pp.128-137
- /Mac82/ Mackert, L.; Keramidis, S.:
A Formal Method for Specifying Hardware Modules
EUROMICRO, 1982, pp.229 - 239
- /Mae77/ Maehle, E:
Fehlertolerante Rechnerstrukturen
IMMD-Berichte, 10(4), 1977

- /Mel74/ Mell, W.D.; Preus, P.; Sandner, P.:
Einführung in die Programmiersprache PL/1
BI Hochschultaschenbücher, 1974
- /Mer77/ Merlin, P.M.; Randell, B.:
State Restoration in Distributed Systems
FTCS 8, 1977, pp. 129 - 134
- /Min82/ Minoura, T.; Wiederhold, G.:
Resilient Extended True-Copy Token Scheme for
a Distributed Database System
IEEE Trans. SwE, 8, 1982, pp. 173 - 189
- /Neu80/ Neudorfer, E.; Schmidt, G.; Sendler, W.:
Möglichkeiten der Fehlertolerierung und Pro-
jektierung der Zuverlässigkeit bei dezentra-
len Prozeßautomatisierungssystemen
Fachberichte MSR, Band 5, Springer Verlag,
1980, pp. 517 - 541
- /Par75/ Parnas, D.L.:
The Influence of Software Structure on Re-
liability
ACM SIGPLAN Notices, 10(6), 1976, pp.358-362
- /Pat81/ Patz, M.:
Zustandssicherungs- und Wiederanlaufverfahren
für verteilte Netzbetriebssysteme mit dyna-
mischer Redundanz
Dissertation, TU München, 1981
- /Pra80/ Pradhan, D.K.; Stiffler, J.J.:
Error-Correcting Codes and Self-Checking Cir-
cuits, COMPUTER, 3, 1980, pp. 27 - 38

- /Ram79/ Ramamoorthy, C.V.; Bastani, F.B.; et.al.:
A Systematic Approach to the Development and
Validation of Critical Software for Nuclear
Power Plants
IEEE Conf. SwE, 1979, pp. 231 - 240
- /Ram80/ Ramamoorthy, C.V.; Bastani, F.B.:
Practical Considerations for the Development
of Process Control Software
Fachberichte MSR, Band 5, Springer Verlag,
1980, pp. 460 - 516
- /Ram81/ Ramamoorthy, C.V.; Mok, Y.R.; et. al.:
Application of a Methology for the Develop-
ment and Validation of Reliable Process Con-
trol Software
IEEE Trans. SwE, 7, 1981, pp. 537 - 555
- /Ran75/ Randell, B.:
System Structure for Software Fault Tolerance
IEEE Trans SwE, 1, 1975, pp. 220 - 232
- /Ran79/ Randell, B.:
Reliable Computing Systems
In: Operating Systems - An Advanced Course
Springer Verlag, 1979, pp. 282 - 391
- /Reu80/ Reuter A.:
A Fast Transaction Oriented Logging Scheme
for UNDO-Recovery
IEEE Trans. SwE, 6, 1980, pp. 348 - 356
- /Reu81/ Reuter A.:
Fehlerbehandlung in Datenbank-Systemen
Hanser-Verlag, 1981

- /Ros77/ Ross, D.T; Schoman, K.E.:
Structured Analysis for Requirements Definition
IEEE Trans. SwE, 3, pp. 6 - 15
- /Rou77/ Roubine, O.; Robinson, L.:
SPECIAL Reference Manual
SRI International, 1977
- /Rus75/ Russel, D.L.; Bredt, T.H.:
Error Resynchronisation in Producer-Consumer-
Systems
ACM Operating Systems Review, 1975,
pp. 106 - 113
- /Rus80/ Russel, D.L.:
State Restoration in Systems of Communicating
Processes
IEEE Trans. SwE, 6, 1980, pp. 183 - 194
- /Schn80/ Schneider, H.J.:
Software - Technologie
Vorlesung Universität Erlangen, 1980
- /Schu80/ Schulz, A.:
Software-Entwurfsmethoden: Ein Vergleich
Angewandte Informatik, 9, 1980, pp. 371-379
- /Shr81/ Shrivastava, S.K.:
Structuring Distributed Systems for Recover-
ability and Crash Resistance
IEEE Trans. SwE, 7, 1981, pp. 436 - 447

- /Sne79/ Sneed, H.:
Programmentwurfs-Methoden im Vergleich
Online-ADL-Nachrichten, 9, 1979, pp. 716-721
- /Tan77/ Tandem 16
Guardian (TM) Operating System
Programming Manual
Tandem Computers Inc. 1977
- /Tay80a/ Taylor, D.J.; Morgan, D.E.; Black J.P.:
Redundancy in Data Structures:
Improving Software Fault Tolerance
IEEE Trans. SwE, 6, 1980, pp. 585 - 594
- /Tay80b/ Taylor, D.J.; Morgan, D.E.; Black J.P.:
Redundancy in Data Structures:
Some Theoretical Results
IEEE Trans. SwE, 6, 1980, pp. 595 - 602
- /Tei77/ Teichroew, D.; Hershey, E.A.:
PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems
IEEE Trans. SwE, 3, pp. 41 - 48
- /Tra80/ Trauboth, H.:
Gesichtspunkte und Verfahren zum Erreichen hoher Zuverlässigkeit von Software
Fachberichte MSR, Band 5, Springer Verlag, 1980, pp. 300 - 328
- /Ver78/ Verhofstad, S.S.M.:
Recovery Techniques for Database Systems
ACM Computing Surveys, 10(2), pp. 167 - 195

- /Wed76/ Wedekind, H.; Härder, T.:
Datenbanksysteme II
Reihe Informatik, Band 18, BI, 1976
- /Weg79/ Wegner, P. (ed.):
Research Directions in Software Technology
MIT Press, 1979
- /Wür77/ Würges, H.:
Unerwünschte Ereignisse in Software Systemen
Informatik Fachberichte GI, 7. Jahrestagung,
Springer Verlag, 1977, pp. 75 - 93

Lebenslauf

Geboren: 24. April 1954 in Plattling

Eltern: Georg und Magdalena Weber

Volksschule: 1960 - 1968 in Otzing

Berufsausbildung: 1968 - 1971 als Flurbereinigungs-
techniker in Landau a.d. Isar

Berufsaufbauschule: 1971 - 1972 in Deggendorf

Berufsoberschule: 1972 - 1974 in Regensburg

Fachgebundene
Hochschulreife: 15. Juli 1974

Wehrdienst: 1974 - 1975 in Straubing

Studium: 1975 - 1980 an der Friedrich-
Alexander-Universität Erlangen-
Nürnberg

Studienrichtung: Informatik

Berufliche Tätigkeit: seit 1. November 1980 bei der
Siemens AG, Unternehmensbereich
Energietechnik

gez. Karl Weber

In letzter Zeit sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen:

=====

Band 13

=====

- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M.,
Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen
Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch
organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G. Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die
syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen
ADA und PEARL.
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik.
(März 1981)

- Nr. 1 Pattern Recognition
(Research at Lehrstuhl für Informatik 5)
(Februar 1981)
- Nr. 2 Bunke, H.:
Analyse elektrischer Schaltpläne
(März 1981)
- Nr. 3 Linster, C.-U.:
SYMPOS/UNIX - Ein Betriebssystem für homogene
Polyprozessorsysteme
(Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolch, G.:
Analytic Solution Techniques for Queueing Network
Models of Computer Systems
(Juli 1981)
- Nr. 5 Lindstedt, W.G.:
Ein Verfahren zur Erstellung portabler algorithmischer Laufzeitprogramme
(Juli 1981)
- Nr. 6 Vollmar, R.:
Zum Begriff des Parallelismus bei Polyautomaten
(Juli 1981)
- Nr. 7 Luft, A.L.:
Rationaler Sprachgebrauch und orthosprachliche
Standardisierung als Grundlagen zuverlässiger Software-
Entwicklung und -Dokumentation
(Januar 1982)

Band 14 Fortsetzung

=====

Nr. 8 Händler, W.:

PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
ARCHITECTURE

(Februar 1982)

Nr. 9 Beth, Th., Hess, P., Wirl, K.:

Materialien zur Kryptographie

(März 1982)

Nr. 10 Jahresbericht 1981 der Informatik

(März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)

- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr.10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)
- Nr.11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr.12 Brendel,W.,Fan,Z.,Klar,R.,Schmielau,W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr.13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozess-Programmiersprachen
(Oktober 1982)
- Nr.14 Sommer,U.,Steinchen,J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr.15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)

- Nr. 16 Bolch, G., Bruchner, W.:
 Analytische Modelle für symmetrische Mehrprozessor-
 anlagen mit dynamischen Prioritäten
 (Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
 (März 1983)

- Nr. 1 Gall, R.:
Formale Beschreibung des inkrementellen Programmierens-
im-Großen mit Graph-Grammatiken
(März 1983)
- Nr. 2 König, R.:
Beiträge zur algebraischen Theorie der formalen Sprachen
(März 1983)
- Nr. 3 Welbourn, D.B.:
The Design of Mechanical Components and the Development
of DUCT:
17 Years of CAD/CAM in Cambridge University
(April 1983)
- Nr. 4 Weitok, R.:
Die System-Implementierungs-Sprache ALICE
(Mai 1983)
- Nr. 5 Schneider, S., Schwab, P., Renninger, W.:
Wesen, Vergleich und Stand von Software zur Produktion
von Systemen der computergestützten Unternehmensplanung
(Juni 1983)
- Nr. 6 Händler, Herzog, Hofmann, Schneider:
Projektabschlußbericht 1978 - 82 EGPA
Erlangen General Purpose Array
(Juli 1983)
- Nr. 7 Mackert, L:
Modellierung, Spezifikation und korrekte Realis-
sierung von asynchronen Systemen
(Juli 1983)
- Nr. 8 Reitenspieß, M.:
Sprachkonstrukte zur Spezifikation und korrekten
Implementierung von Schutzproblemen
(Juli 1983)

